2009

Fable: Finite automata based learning engine

Ryan Michael Jackson

FABLE: FINITE AUTOMATA BASED LEARNING ENGINE

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Ryan Michael Jackson

June 2009

FABLE: FINITE AUTOMATA BASED LEARNING ENGINE

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Ryan Michael Jackson

June 2009

Approved by:

[redacted]

George M. Georgiou, Chair, Department of Computer Science and Engineering

6-8-2009

Date

[redacted]

Kerstin Voigt

[redacted]

Ernesto Gomez

ABSTRACT

FABLE, a finite automata based learning engine, is an attempt to create a system that is capable of learning how to solve problems without any prior knowledge about them. It accomplishes this by building increasingly more accurate FA models of the problem. FABLE uses the FA models to make decisions that are more likely to lead to a desired goal state. The results of each choice are recorded and used to improve the FA model for future use. Current research into the automated building of FA models have been focused on creating algorithms that learn to recognize the same strings as a target FA with an accuracy that is PAC or probably approximately correct. For these algorithms, the building of the FA is the goal itself. FA based decision systems have been created for solving problems, but in most cases the FA has been designed, not learned automatically. FABLE attempts to join these two areas together by using automated building of FA models for the purpose of solving specific problems. In particular, FABLE is tested on the problems of minesweeper, tic-tac-toe, and checkers. The results show that in all three cases, FABLE learned to play significantly better than the results expected from random play within less than 100 games.

ACKNOWLEDGEMENTS

I wish to express my appreciation to all those who have helped me in this project. I greatly appreciate the help and encouragement that I have received from Dr. Georgiou. My thanks go to Dr. Voigt and Dr. Gomez for their input and willingness to help. I would also like to thank my family for all their support. I especially want to thank my wife, Tamara, for her patience and understanding while I worked on this project.

DEDICATION

To Tamara and Scott for all of their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. BACKGROUND

1.1 Introduction

In this chapter the purpose of this thesis will be explained. Once the purpose has been established, an analysis of the nature of the problem will be considered. The significance of this thesis in relation to the problem will then be presented, followed by a definition of terms used in this thesis. Hypotheses regarding the efficacy of the proposed solution will be discussed, and then the scope and limitations of the current research will be put forward. Finally, a brief overview of the organization of the remainder of the thesis will be given.

1.2 Purpose of the Thesis

The purpose of this thesis is to investigate the use of finite automata based modeling as a method of pursuing artificial intelligence. In particular, the issue of intractability of such methods when applied to all but the simplest of problems will be considered. The problem of intractability will be handled through the Finite Automata Based Learning Engine (FABLE) which was created specifically for this purpose. The method by which FABLE will avoid intractability will be by deploying multiple heuristics to the problem, with the hope that at least one of the heuristics will be able to build a good approximation within a reasonable amount of time.

1.3 Nature of the Problem

The nature of the problem lies in determining how to build a finite automaton in a reasonable amount of time that allows desired outcomes of a task to be achieved. This is difficult to achieve in practice because any problem that is sufficiently complicated to be interesting will generally also require a generous number of states to model effectively. It is also likely that the size of the alphabet of the finite automaton will be non-trivial. Since the possible number of configurations grows exponentially as the number of states increases, a brute force approach can quickly become intractable for all but the simplest of problems. That is not to say that brute force can not work. A non-FA based brute force approach to solving the game of checkers was recently accomplished by Schaeffer et al. This was accomplished despite the fact that "[t]he game of checkers has roughly 500 billion billion possible positions." [18] Even with this recent victory for brute force techniques, it is still a very inefficient learning method that should be avoided when possible.

1.4 Significance of the Thesis

The significance of this thesis will be to demonstrate the feasibility of building a system that is capable of automatically learning how to increase performance at a task that is presented to it by means of modeling the problem with a finite automaton. Current work in the field of automatic learning of FA's is mostly focused on the building of the FA itself. This thesis will instead focus on the automated building of FA models to allow better decisions to be made in the future. In particular, it will focus on demonstrating the feasibility of using such methods for improving the

decisions of an agent presented with the problem of playing a board game without any prior knowledge about the game.

1.5 Definition of Terms

The following is a list of terms and their definitions as they pertain to this thesis.

FA - Short for finite automata or finite automaton. A finite automaton is a computing machine which consists of a collection of states, a defined alphabet of valid inputs, and a set of transition functions mapping one state to another based on the input received.

DFA - Short for deterministic finite automata. A deterministic finite automaton is a special case of FA in which the set of transitions has been limited to only permit one transition per symbol in the alphabet per state. That is, there is never any ambiguity as to which state to transition to on any input.

NFA - Short for non-deterministic finite automata. A non-deterministic finite automaton is an FA where there is no restriction on the number of transitions that can exist for any state given the same input.

PFA - Short for probabilistic finite automata. A probabilistic finite automaton is a NFA where there are specific probabilities as to which transition will be taken for a given state with more than one defined transition for the given input. For example, if a state called s1 has two transitions for the input a and the first transition has a probability of $\frac{2}{3}$ and the second a probability of $\frac{1}{3}$, then the first transition will be taken twice as often as the second transition.

MAT - Short for minimally adequate teacher. The minimally adequate teacher is a teacher that helps a learner to construct a target FA by answering questions from the learner. To be minimally adequate, it is required that the teacher already know the target FA. In addition, the teacher must always truthfully answer membership and equivalence queries. The teacher must also provide counterexamples as necessary.

PAC - Short for probably approximate correct. When a learner is attempting to construct a target FA and there is no access to some sort of equivalence query to verify that its current model matches the target FA, the learner may have to settle for a FA that is PAC. This simply means that the constructed FA generates accurate results from all known inputs and appears to make accurate predictions when tested with inputs not seen before. It is still possible that there is a counterexample input to the constructed FA, but we can say that the constructed FA is PAC.

Game Tree Size - For any problem where there is a clear start state and the eventuality of reaching a clear end state is guaranteed, the game tree size is measured by looking at all branches from the initial state and traversing them until all possible leaf nodes have been counted. That is, the game tree size of a problem is not just a measure.of all the possible end states of a task, but of all the possible ways in which each end state can be reached.

State Space Complexity - For any problem where there are clear states and transitions between them, the State Space Complexity of the problem is the sum of all possible states. This differs from the game tree size in that it counts the total number of possible states, without concern for how the state was reached.

Heuristic - For the purposes of this thesis, a heuristic will be any form of pattern recognition algorithm that can be added to the basic bottom-up learning process to promote increased efficiency of learning. An example of a heuristic is the nearest neighbor algorithm. The nearest neighbor heuristic makes the assumption that when a new state or input is encountered, that the new state or input will act most similarly to the known states or inputs that are closest to it. Such a heuristic will likely yield good results for character recognition, but could be disastrous if applied to chess.

FABLE - Short for finite automata based learning engine. FABLE is the acronym for the system that was developed for the purpose of testing the hypothesis of this thesis. A basic description of FABLE is that it is a bottom-up learning engine that automatically constructs a finite automaton to match its observed reality by recording its current state, the input received, and the state that resulted from that input. Since a finite automaton is easy to traverse and understand, it is also possible to hardcode states and transitions into the automaton in order to facilitate the encoding of top-down knowledge. In addition, it is possible to incorporate heuristics into FABLE that attempt to make predictions or fill in the gaps for those states and inputs which have not yet been directly observed or encoded.

1.6 Hypothesis

The basic hypothesis of this thesis is that a system can be built which learns by observation and stores the acquired knowledge in the form of a FA which is used to guide future decisions. In particular, it is hypothesized that this approach will be effective at learning to play various board games without the need to create any

code that is specific to any of those games. An additional constraint is that both the learning and decision making processes must not exceed polynomial time complexity.

1.7 Scope and Limitations

In order to test the hypothesis, the Finite Automata Based Learning Engine, hereafter referred to as FABLE, will be created. It will be used to test the hypothesis by implementing a system of the type referred to in the hypothesis on 3 board games. These games are minesweeper, tic-tac-toe, and checkers.

In each case, the capability to play the game on boards of various size will be implemented. As the size of the board is increased, so will the number of possible states and the corresponding number of possible transitions that will need to be learned in order to be an effective player of these games.

The FABLE engine will be designed to be able to make use of several heuristics. The design will also be such that new heuristics can be added as necessary. For the current implementation of FABLE the following heuristics will be implemented: the observed state heuristic, the nearest neighbor heuristic, the locality heuristic, and the rotationally invariant heuristic.

1.8 Organization of the Thesis

This thesis is divided into 5 chapters. The first chapter covers an introduction followed by a brief description of the purpose of the thesis and an overview of the nature of the problem. The significance of the thesis is then discussed, followed by a list of terms used within this thesis. A hypothesis is then given and the scope and limitations of

the work in this thesis related to that hypothesis are then explained. Chapter 1 is then concluded with an overview of the organization of the thesis as a whole. Chapter 2 is a review of previous literature as it pertains to this thesis. Chapter 3 covers the methodology used to test the hypothesis of this thesis. This includes a description of the FABLE system and how it works. Once FABLE has been described the basic methods whereby the system will be evaluated are given. Chapter 4 covers the actual results that were generated by the FABLE system and an analysis of those results is then done to verify if the actual results match with the expected results. Chapter 5 will discuss how the results validate the thesis and suggest areas for further research that would validate it further. Following Chapter 5 there will be a list of the relevant references.

2. LITERATURE REVIEW

2.1 Introduction

Chapter 2 is a review of the relevant literature to put the work of this thesis into its proper context. In particular, literature regarding methods for building a FA will be reviewed. This field of inquiry is usually sub-divided into the branches of learning with and without a teacher. Literature regarding both approaches will be reviewed. After this, literature dealing with the use of FA for the purpose of making intelligent choices will be reviewed.

As a side note, within the broader area of artificial intelligence, it is more customary to refer to these two learning methods as supervised and unsupervised learning. However, in the specific field of automatic FA construction, sometimes known as grammar inference, it is usually referred to as learning with and without a teacher. Since FABLE is based on the building of FA automatically, the nomenclature within this thesis will use the terminology of learning with or without a teacher.

2.2 Automatic Construction of Finite Automata with a Teacher

Several methods have been developed for constructing a FA with varying degrees of efficiency. One of the earliest results into investigating this problem was found by E. Mark Gold who showed that the time complexity of learning consistent FAs from

given data is computationally intractable [12]. Since this finding, the majority of efforts to construct FAs have done so from the perspective of having more than just the given data to work with.

The most commonly referred to algorithm that resolves the intractability issue for the problem of constructing a FA is that developed by Angluin in "Learning Regular Sets from Queries and Counterexamples" [3]. In this paper Angluin introduced the concept of the minimally adequate teacher or MAT. The basic process described here consists of two main operations in which the learner submits queries to the teacher. The first type of query is a membership query where the learner generates a string and asks the teacher if the string is contained in the target FA. The teacher will respond with either a yes or no. The second type of query is the equivalence query. In this, the learner submits what it believes to be the target FA and the teacher then answers yes or no. If the answer is no, the teacher will then provide a string that is a counterexample to the submitted FA. In this technique, the intractability problem is solved by the use of a teacher which provides more information than just the given data. Using this approach the problem of constructing a FA can be solved in polynomial time.

Since Angluin introduced this approach, several others have modified the algorithm in order speed up the process or to make the process easier to understand. Parallelizing the algorithm was one of the techniques tried to reduce the time it takes to learn. Balcazar et al [4] found that a DFA can be constructed in O(n/log n) provided that there are polynomially many processors available to distribute the problem over. Yokomori looked at modifying Angluin's basic algorithm which was designed

for DFA's and adapted it to learning NFA's [22].

While Angluin's algorithm and its variants are interesting to look at due to their efficiency, they do have a significant drawback in regards to their use for solving the problem of constructing a FA in general. The problem is that in order for the algorithm to work, the target FA must already be known by the teacher and the teacher must always answer membership and equivalence queries made to it by the learner. This algorithm therefore does not help in the event that the target FA is unknown.

2.3 Automatic Construction of Finite Automata without a Teacher

Now that the problem of learning a FA where a MAT is available has been demonstrated to be tractable, let us consider the case where a MAT is not available. For a case such as this we can consider a few different possibilities.

To begin, let us consider that the absence of a teacher does not imply that no questions can be asked. Let us suppose that the target FA has been implemented and is concealed within a black box that accepts inputs and gives outputs. By sending inputs into the black box and observing the corresponding outputs, it may be possible to determine what the FA inside the black box is doing. In this case, although no equivalence queries are possible, the equivalent of membership queries can still be asked. This is a harder problem to solve, and it would be impossible to ever know for sure that the constructed FA is an exact match to the target FA, but it would potentially be possible to construct a FA that gives accurate predictions over a wide range of inputs. A FA that is an effective model for an unknown FA is often referred

to as being PAC. In most cases, when learning a FA without a teacher, the goal is merely to construct a FA that is PAC.

2.3.1 *Constructing Finite Automata without a Teacher using Passive Learning*

Attempts to determine the unknown FA where the learning agent has no access to control the input to the unknown FA are referred to as passive learning. In "Efficient Learning of Typical Finite Automata from Random Walks", Freund, et al [11] found that passive learning is capable of learning a PAC model of the unknown FA with only an expected polynomial number of mistakes given the constraint that the target FA was generated randomly or at least semi-randomly. Note that this does not state that it will learn with only a polynomial number of mistakes, the worst case scenario is still an intractable exponential number of mistakes. Freund, et al merely established that the average case will be polynomial and therefore tractable on average.

An excellent survey paper covering the problem of passively learning DFA was put together by Cicchello and Kremer in "Inducing Grammars from Sparse Data Sets: A Survey of Algorithms and Results" [8]. In this paper, a summary of several algorithms relating to this problem were summarized. These algorithms included SAGE [13], Ed-beam [15], TBW-EDSM [8], MMM [17], and Exbar [15]. Each of these algorithms makes different trade-offs between accuracy of the generated FA, the amount of computation required to generate the FA, and the robustness of what type of FA the algorithm can learn. For example, the MMM algorithm attempts to construct a perfect replica of the target FA and is computationally expensive. TBW-EDSM on the other hand does not aim for a perfect match, but one which will give

results that are accurate with about 99% probability. The final result of TBW-EDSM may not be quite as accurate as the result of MMM, but the result can be found faster.

2.3.2 *Constructing Finite Automata without a Teacher using Active Learning*

Attempts to determine the unknown FA where the learning agent can control or at least influence the input to the unknown FA are referred to as active learning. Algorithms for active learning are often very similar to passive learning algorithms with the exception that in addition to the algorithm that constructs the FA based on input and output, it also needs an algorithm for deciding what input to attempt next.

The basic motivation for allowing the learning agent to directly control the input is to allow it the opportunity to try to generate output that will fill in the gaps in its current model or to test its current model specifically in those areas which are under the most suspicion of being incorrect. Naturally, the expectation is that since the learner can now actively test its model, the accuracy of the resulting model will be higher or found in less time than when only random data is made available to it. Indeed this is the case, as demonstrated by Bongard and Lipson [6] who utilize an evolutionary learning algorithm that can actively select inputs. They find that having this ability allows their algorithm to outperform the EDSM algorithm for passive learning not only in speed, but also in the types of FA that the algorithm can learn without the problem becoming intractable. In particular, Bongard and Lipson show that their algorithm works well even for FAs that are not as closely balanced as the passive learning algorithms require. That is, it can perform adequately even when

the target FA does not have a roughly equal number of accepting and non-accepting states with branches that are mostly the same in depth. Effectively, active learning permits polynomial time learning even for FA that stray somewhat from the expected structure of a randomly generated FA. Despite this gain in general learning capability compared with purely passive forms, the problem of learning a target FA still remains intractable in the truly general sense.

2.4 Literature on the Use of Finite Automata for Intelligent Agents

The use of FA for intelligent agents has been done in several different ways. These range from FA that were specifically engineered to perform a certain task, to applications where the FA was built using input from human experts to generate an FA using some of the algorithms mentioned in the previous sections.

Several real world problems have made use of FA based solutions where the FA was designed from a top-down level. Fregene, et al, showed that FA based decision making could be used to create the controls systems for a multi-agent system that would work together to control unmanned ground vehicles for the purpose of creating maps [1]. Feiliang used FA as part of a language translation system to automatically translate text between Chinese and Japanese [9]. FA have even been used to control animated characters in an animation engine by Martins [16].

Real world problems have also been solved by FA where the FA is built using inference methods. In "An Automata-based Approach to Robotic Map Learning", Basye detailed a method in which a robot built a map of its surroundings with the map being learned as a FA [5]. The robot could then use the FA to navigate from

point to point.

2.5 Summary

The purpose of this thesis is to use a constructed FA for the purpose of making intelligent decisions. Due to this, it is implied that the agent will have at least some capability to influence the input going in. One key difference between the goal of learning FA for this thesis as opposed to the current body of work regarding learning FA is that most research into this field has been focused on attempting to learn the entirety, or at least a very close approximation, of the target FA. In this thesis the goal is merely to achieve a good enough model to allow an agent to achieve putting the target machine into a desired state. This may not actually require a total knowledge of the entire FA. In fact, it may only require a small fraction of the target FA.

An example of this type of problem can found in the game of checkers. Checkers can clearly be modeled as a DFA, but with its 500 billion billion possible states [1] the complete DFA is still unknown. In addition, the learning algorithms found to date require at least the equivalent of $O(n^2)$ computations to create the model using a MAT. Without a MAT, the computational requirements would be at least $O(n^3)$ if the DFA happens be well-balanced, otherwise it will be exponential. Since the complete DFA for checkers is unknown, a MAT can not be used. Despite this, it is easy enough to test if a given sequence of moves was valid in a game of checkers. It is also easy to determine what the end state of a valid game was. Due to this, active learning is possible to use with the problem of checkers. However, substituting 500 billion billion for n in an algorithm that runs with a time complexity of at least $O(n^3)$

results in such a large amount of computation that it is effectively intractable with current hardware and algorithms.

Despite this apparent intractability for large problems, this thesis contends that most board games have patterns to them that allow FA models that are good enough to enable decisions that are better than making choices at random to be constructed in polynomial time. For this to work, the pattern would need to be one that can be found by one of the heuristics in the system. Furthermore, FABLE does not necessarily seek to learn a FA that is a complete model of the problem in its entirety. For practical purposes, it only needs to learn enough of the FA that models the problem to allow whichever heuristic(s) apply to the current problem to do a good job of filling in the gaps. This will then allow the agent to decide on supplying inputs that are more likely to result in states that the agent desires than states which the agent does not desire. In more precise terms, intelligence of some sort will be said to have been achieved when the FABLE agent is capable of consistently outperforming an agent where all choices are made on a purely random basis. No claim of completeness of the model or of the optimality of the choices made is implied. The objective is to demonstrate better decisions than picking at random within a tractable amount of time and training.

3. METHODOLOGY

3.1 Introduction

This chapter will explain the creation of the FABLE system for dealing with the problem. It will discuss the design requirements for the FABLE system, give an overview of the design, discuss the implementation of the system, and then summarize.

3.2 Design Requirements

3.2.1 Game Rules and Parameters

The FABLE system will be tested using the games of minesweeper, tic-tac-toe and checkers. In order to perform this testing, a game engine will be built that implements these games so that FABLE may interact with them. As the design and implementation of these games is not the focus of this thesis or an item of interest from a research perspective, the discussion of the design of the game engine in this section will be primarily to establish the particulars of the rules and parameters for these games as they will be implemented for testing FABLE.

These games have been selected because each has a start state, a clear mechanism for determining the next state based on the decision a player makes, and a clear condition for the end state. The state of the game resulting after a decision is made

can be used to build a transition that maps from the previous state to the next state based on the selected input that resulted in the transition occurring. For any fixed size grid or board, the number of possible transitions in each of these games is finite. The result is that for any fixed size grid, given sufficient examples, a DFA that perfectly encapsulates the entire game can be constructed. For practical purposes, building a complete DFA will become intractable as the size of the grid or board increases. To deal with this problem, several heuristics will be built that attempt to fill in the gaps of what is known. Using heuristics it should then be possible to make intelligent decisions even in situations which have not been encountered before. These heuristics will be discussed later in this chapter in section 3.3.3.

Minesweeper

Minesweeper is a single player game with clear conditions for a win versus a loss. Essentially, minesweeper is comprised of a grid in which a known number of mines have been placed at random. Initially, the player has no information whatsoever about the position of those mines. To play, the player selects a square in the grid to be clicked. If the square in question is the location of a mine, then the game ends with a loss. Otherwise, the square will change from being blank to displaying a number corresponding to the number of mines that are directly adjacent to that square in the grid. If the number is 0, it will cause all adjacent neighboring cells to be clicked automatically. To win, a player must click all of the cells in the grid which are not the location of a mine.

For those that are familiar with the popular implementation of minesweeper found

on almost every copy of the Windows operating system, it is important to note the game engine for this thesis will follow the rules as stated above. Unlike the Windows version, the version in the game engine will arrange the mines at the beginning of the game. This means that it is possible to lose on the very first click.

From personal experience during the testing phase of the game engine, I found that the mine density was a very important consideration of the game. In general, a higher density of mines on the grid made the game harder while a smaller number made it easier. I decided to have the game engine use a mine density of 20%. This value was chosen as I found it to a be a density that is high enough to make winning a non-trivial process, but also not so high that it makes winning virtually impossible no matter what strategy is used.

Tic-Tac-Toe

Tic-Tac-Toe is a very old game which almost everyone knows how to play. In its usual form it consists of a 3x3 sized grid or board which is initially blank. It is a two player game with the first player normally being designated as X and the second player designated as O. Play starts with player X choosing any of the squares on the board and placing an X inside the square. Player O then takes a turn by placing an O in any of the remaining squares. Play continues in this manner with player X and player O alternating turns to place their respective letter in one of the available squares. To win, a player must place their letter in 3 cells which comprise a row along a vertical, horizontal, or diagonal line. The game terminates when all of the squares have been filled or one of the players wins, whichever comes first.

The rules within the game engine for this thesis have been modified slightly from the standard rules. Instead of completing 3 in a row being the victory condition, it will award 1 point for each such row. Due to this change, play will continue until all squares on the board have been filled. Once the board is completely filled, the player with the most points wins. If the points are the same, the game is a tie. These changes were made so that when larger board sizes are considered, the game will remain interesting. Without these changes it is trivial for player X to win every time for any board where the height and width are larger than 3. This is due to the fact that player X will always be able to create two in a row on the second turn such that 3 in a row can be completed on the third turn from either side of the 2. Since player O can only block one of the sides, player X simply has to place an X on the third turn that completes 3 in a row on the side that player O did not block.

Checkers

Of the games that FABLE will be tested on, checkers is the most complicated. Rather than write the rules to the game in my own words, I have decided to include the official rules as given by the American Checkers Federation. The rules as found below are taken directly from the American Checkers Federation without any modification. The are 16 rules in total:

> Rule 1 - The official checkerboard to be used in national tournaments and official matches shall be green and buff with two-inch squares. The board shall be placed for playing so that the green double corners are on the right-hand side of the players.

Rule 2 - The official checkers to be used in national tournaments and official matches shall be turned and round, red and white, and of a diameter not less than one and one-quarter inches nor more than one and one-half inches. The pieces shall be placed on the green squares.

Rule 3 - At the beginning of the contest, the players shall toss for colors. The first move is made by the player having the red checkers. Thereafter, the players shall alternate in leading off with red in each succeeding opening balloted.

Rule 4 - Each player must make thirty moves per hour. Upon completion of each set of thirty moves, the player completing the thirty moves adds one hour to his/her clock. If a player's time expires before he/she makes the allotted thirty moves, that player forfeits the game, and his/her opponent is declared the winner by default.

Rule 5 - The men can move only diagonally forward, one space at a time, to an unoccupied adjacent dark square in the row ahead. If there is an opponent's checker or king on an adjacent square in the row ahead with an unoccupied square on the same diagonal line in the following row, that piece (checker or king) must be jumped. Checkers and kings cannot jump over their own pieces.

Rule 6 - When there are two or more ways to jump, five minutes shall be allowed for the move. When there is only one way to jump, time shall be called at the end of one minute, and if the move is not completed at the end of another minute, the game shall be adjudged as lost through improper

delay.

Rule 7 - If a player has more than one way to jump, he may select whichever one he wants, regardless of the number and type of pieces that can be captured.

Rule 8 - At the beginning of a game, each player shall be entitled to arrange his own or his opponent's pieces properly on the squares. After the game has opened (a move has been made), if either player touches or arranges any piece without giving intimation, he shall be cautioned for the first offense and shall forfeit the game for any subsequent offense of this kind. If a person whose turn it is to play touches one of his playable pieces, he must either play it or forfeit the game.

Rule 9 - If any part of a playable piece is played over an angle of the square on which it is stationed, the play must be completed in that direction. Inadverdently [*sic*] removing, touching, or disturbing from its position a piece that is not playable, while in the act of jumping or making an intended move, does not constitute a move, and the piece or pieces shall be placed back in position and the game continued.

Rule 10 - When a checker reaches the crownhead of the board by reason of a move or as the completion of a jump, it becomes a king. That completes the move or jump. The checker must then be crowned by the opponent by placing a piece on top of it. If the opponent neglects to do so and makes a play, then any such play shall be put back until the piece that should have been crowned is crowned. Time does not start on the player whose piece

should have been crowned until the piece is crowned.

Rule 11 - A king once crowned can move in any direction as the limits of the board permit. A king can jump one or more pieces in any diagonal direction as the limits of the board permit. When a piece is not available for crowning, one must be furnished by the referee.

Rule 12 - A draw is declared when neither player can force a win. When one side appears stronger than the other, and the player with what appears to be the weaker side requests the referee for a count on moves, then, if the referee so decides, the stronger party is required to complete the win, or to show to the satisfaction of the referee at least an "increased" advantage over his opponent within 40 of his own moves, these to be counted from the point at which notice was given by the referee. If he fails to do this, he must relinquish the game as a draw.

Rule 13 - After an opening is balloted, neither player shall leave the board without permission of the referee. If permission is granted to a player, his opponent may accompany him, or the referee may designate a person to accompany him. Time shall be deducted accordingly from the player whose turn it is to move.

Rule 14 - Anything that may tend to annoy or distract the attention of an opponent is strictly forbidden, such as making signs or sounds, pointing or hovering over the board either with the hands or the head, or unnecessarily delaying to move a touched piece. Any principal so acting, after having been warned of the consequences and requested to desist, shall forfeit the

game.

Rule 15 - Players shall be allowed to smoke during the conduct of a game, but care must be exercised not to blow smoke across the board, lest it annoy an opponent. If a player is thus annoyed, he may object to his opponent smoking, in which case neither player shall be allowed to smoke.

Rule 16 - Any spectator giving warning either by signs or sound or remark on any of the games, whether playing or pending, shall be ordered from the room during the contest. Play shall be discontinued until such offending party retires. Spectators shall not be allowed to smoke or talk near the playing boards [21].

Barring the rules that are obviously not applicable for a computer program, such as the rules regarding whether a player is allowed to smoke, the rules for checkers were implemented in the game engine according to the official rules. Although checkers is usually played on an 8x8 board, the lack of an official rule specifying the size means that the game engine was designed to accommodate various board sizes without the need to make any changes to the official rules.

It is interesting to note that just as the official rules do not state a board size, they do not state the starting position of the pieces. In fact, the official rules do not even specify how many starting pieces there are for either player. Despite this, the expectations of the American Checkers Federation can be found by examining the game histories that they maintain records of. The federation has the complete game history of several U.S. and World championships [20]. Looking at these game histories I was unable to find a single example of any game that was not played on

an 8x8 board with a starting position with 12 red and 12 white pieces. In addition, I was unable to find any instance in which the 12 pieces for red started in any position other than filling the 4 legal positions for a piece in the bottom 3 rows of the board with the 12 pieces for white occupying the top 3 rows.

Although the number of pieces and starting positions are not specified by the official rules, the implementation in the game engine for FABLE does try to maintain the spirit of the implied rules. The only exception being that it does not enforce an 8x8 board. To do this, the game engine is designed such that for any size board, the number of starting pieces for red and white will be the same. In addition, the starting positions will place the red pieces on the bottom of the board, filling every legal position until a row is filled. Every row will be filled up to but not including the middle row. The same will be done for white with the exception that the rows will be filled starting from the top going down. From testing, it was determined that when this is done on boards with an odd number of rows, it is too easy for the game to end quickly due to insufficient space for any of the pieces to move. To alleviate that problem, a restriction on sizes was placed in the game engine to only allow board sizes where the number of rows and the number of columns are even.

For the purposes of testing FABLE, one rule was added in addition to the official rules. This rule is that in the event that no material change has occurred for the last 50 moves, the game engine will declare the game to be a tie. A material change is defined to be the advancement of a regular checker or the capturing of a piece. The basic idea of this rule is not mine. I found that several variations of this rule are popular among the various free implementations of checkers on the web. In each case,

the addition of a rule similar to the one I used was for the purpose of ensuring that every game does come to an end eventually. My choice of the number 50 is arbitrary, but it seemed like it should be long enough to not severely distort the game results.

3.2.2 Data Structure and Storage Requirements

As the FABLE system is to be a FA based learning engine, the most basic data structure requirement for the system is how the FA will be structured within memory and stored. As this is to be a generic system, no decision can be made for the best structure or storage based on the expected size of the resulting FA. Due to this, the structure and the storage mechanism need to be highly scalable so that they will be both effective as well as efficient as the size of the problem is changed. As there will be multiple heuristics attempting to build a FA that models the problem, the ability to store multiple FA's is also necessary. Good design techniques always promote reusability, and for this reason the storage of each FA should be done in as similar a manner to the other FA's as possible while still maintaining robustness and scalability.

In addition to the structure and storage of the FA's, the system should also incorporate a method of storing the history of actual inputs and outputs that have been observed. Having this history will allow any FA that has been constructed to be tested against the actual history to determine its accuracy and robustness on at least all known data. Since the history of interaction will do nothing but grow over time, there is no upper limit on how large it may become other than the limit of available memory unless an artificial limit is imposed. If such a limit is to be imposed, then an algorithm for determining what history should be deleted would need to be imple-

mented. For the purposes of the current implementation it is assumed that sufficient memory will be available at all times, thereby removing the need to implement any artificial limit. Due to this, the size of the history will certainly grow larger with every interaction. This dictates that the history must be stored and structured in a manner that will allow for efficient data operations despite a very large data set.

3.2.3 Heuristic Design Requirements

The heart of the FABLE system to avoid intractability is the use of heuristics to look for underlying patterns that allow useful models of the problem to be built with less data and processing than brute force methods would require. In order to take better advantage of current hardware capabilities, the ability to make use of multiple cores on the same processor, as well as cores on processors on computers that are connected through a network would also be helpful. To achieve these goals, the implementation of the heuristics needs to be written in a manner that allows them to be distributed. It also requires that the means whereby the heuristic algorithms will access history information, as well as store constructed FA's needs to be done in a manner that will work across a network. The exact manner in which a heuristic will build a FA is dependent on the nature of the type of pattern that the particular heuristic is searching for. Since the goal is to solve problems in a tractable time with a tractable amount of training data, the pattern to be searched for should be one that can be recognized in at most a polynomial time frame.

3.2.4 Agent Design Requirements

As the FABLE system is an approach to AI, it has a need of a system for choosing between different choices. Within FABLE this system well be termed as the agent. The job of the agent will be to look at the current state, analyze the possible decisions it can make, and then make the decision that it believes is most likely to result in the desired outcome. Since the agent will be interacting with the problem directly, it is assumed that the agent can see the current output state of the problem that it is dealing with. For a game such as checkers this would correspond to the current board position. It is also assumed that the agent will know all of the possible actions (inputs to the problem) that it may take. In addition, the agent will need to be assigned a goal state that it seeks to reach. The agent will require an algorithm for assessing the desirability of any possible action. An algorithm for selecting which action to take also needs to be selected. Whatever algorithm is implemented, the decision needs to be made in a reasonable amount of time.

3.3 System Design

3.3.1 System Design Overview

The basic overview of the FABLE system can be visualized with the level 0 data flow diagram in Figure 3.1. As a top level overview of FABLE, it shows that the basic idea is for the game engine, which will be representing the problems to be learned and modeled, will communicate with FABLE by providing selected moves, board positions, and legal moves as appropriate. In turn, FABLE provides selected moves to the game engine when it has the opportunity to do so.

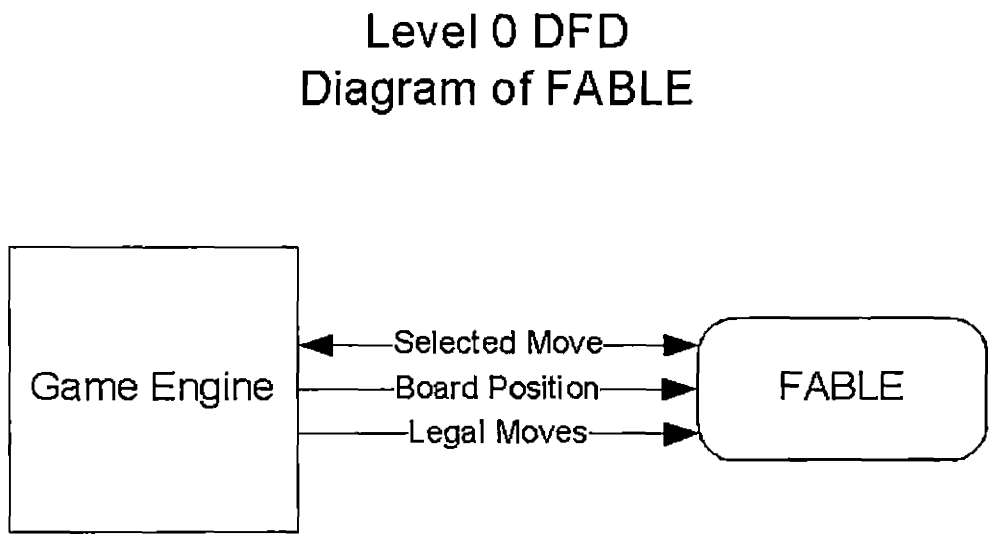

Fig. 3.1: Level 0 Data Flow Diagram Diagram.

Figure 3.2 is a level 1 data flow diagram and reveals a greater level of detail of the workings of FABLE. Now it is seen that for FABLE to work, there will be three main processes at play. The first is the history recorder. The job of the history recorder is to store a complete history of every game that is played on the game engine. The second process is the heuristic FA constructor. In actuality, there are many heuristic FA constructors, one for each heuristic that is implemented. The constructors use the data stored in the game history to construct a FA model based on the particular type of pattern the heuristic is designed to search for. The last process is the agent. The agent's purpose is to select an input, or move, to supply to the game engine. The game engine will then compute the next board position according to the rules of the game being played. The agent may use any method to determine which possible legal move it wants to select. There will be three types of agents used in FABLE. The first is an agent that simply picks any legal move at random. The second is an agent

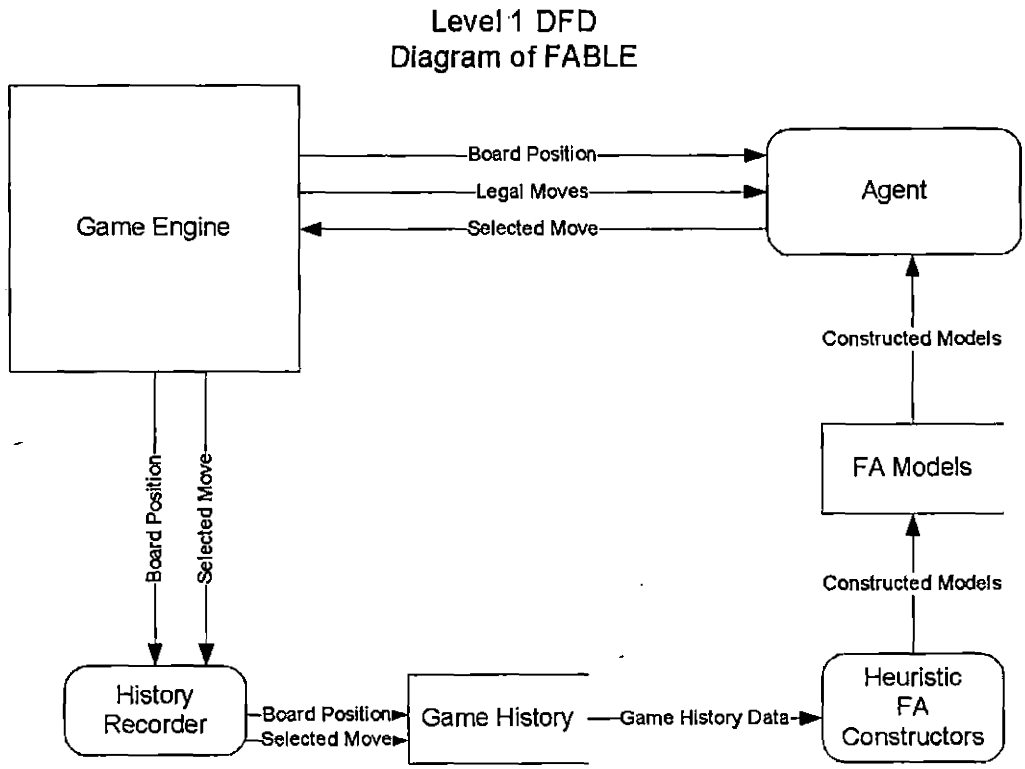

Fig. 3.2: Level 1 Data Flow Diagram Diagram.

that uses the information in the FA models constructed by the heuristics to make an informed decision. The last type of agent is human in the event that a human wants to play. For multi-player games, the agent type for each player can be chosen independently from the agent type for the other players.

3.3.2 Data Structure and Storage Design

In order to meet the design requirements for data storage, the use of an SQL database was selected. This method of data storage and retrieval actually meets the requirements very well. The storage of a FA in a database is accomplished very easily through the use of a table. In addition, the same table can easily be used to store multiple

FA's by simply adding an additional column to the table that identifies which FA the row entry pertains to. With the appropriate indexes, the ability to store FA's in the database should scale very well and provide adequate performance. A database can store the entire history of all interactions as well. Once again, this table should scale well provided that the appropriate indexes are used. The database structure for FABLE is found in Figure 3.3.

The basic structure described in the diagram can be broken down as pertaining to three main processes. The first is the top-level agent process which interacts directly with the game engine. This process writes requests to the Agent_Request table and then monitors the Agent_Response table for responses generated by the heuristic specific agent processes. Each heuristic reverses this by monitoring the Agent_Request table for requests and then writing its response to the Agent_Response table. Heuristics also make use of the transition table for evaluating what the response should be to each request, and are also responsible for updating the transition table based on the history of previously observed outcomes contained in the Game_History and Game_Type_History tables. If applicable, the heuristic can check the Game_Processed table to ensure that only games which it has not already processed to be included in the Transition table are used to update the Transition table. The game engine itself only interacts with the database for the purpose of recording the complete history of every game played using the Game_History and Game_Type_History tables. To better understand this structure, a more detailed explanation of each table and its columns will follow.

Agent_Request - This table is used by the FABLE agent to send a request for

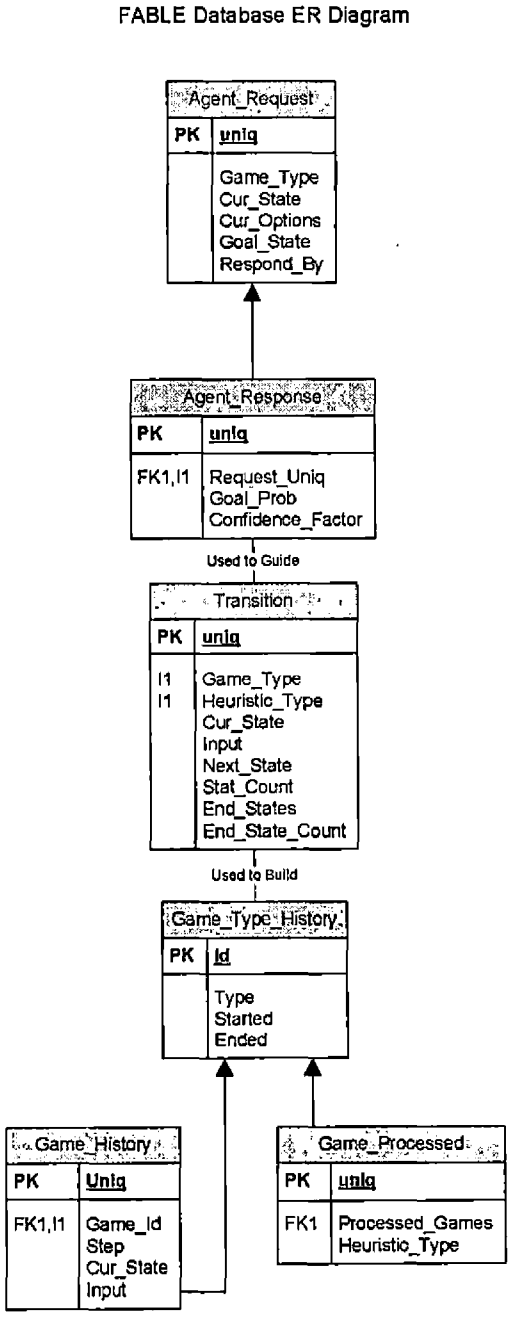

Fig. 3.3: FABLE Database Entity Relationship Diagram.

an analysis of the potential moves. Since FABLE is designed to work in a distributed fashion, processes that build FA's based on particular heuristics will also be responsible for servicing those requests by responding with the probability of any given option leading to the goal state. It determines this based on the model of the heuristic in question. The same process will also respond with the degree of confidence that the model currently predicts each of those probabilities to have.

The first field in the Agent_Request table is uniq. The majority of the tables in FABLE have a uniq field. In each case, the field serves as the unique row identifier and is of the uniqueidentfier type which is a native database type in MS SQL Server. It is also directly supported within the .net framework in that unique keys can be automatically generated. It is also designated as the primary key.

The second field is the Game_Type field. This field has been added for the purpose of identifying to which game type the current row entry belongs. The coding is very straightforward. For checkers, the game type is simply checkers followed by the dimensionality of the board that was used. For example, if a game of checkers on a board that has 4 rows and 6 columns was played, then the entry for this field would be "Checkers 4 * 6." It is done similarly for minesweeper and tic-tac-toe.

The third field is Cur_State. This field represents the currently observed state. For any task that FABLE is asked to learn, the observed state is equivalent to the same information that a human player would have access to if they were playing the game. For the current implementation of FABLE the agent will not be given any additional information that would normally be hidden from a human player.

The fourth field is Cur_Options. This field is used solely to increase the efficiency

of FABLE. The agent will receive a list of all current legal moves for its current position. The game engine for the three games will not permit illegal moves to be taken by any player, whether that player is human or not. In the event that an illegal move is attempted, the game engine will simply not respond until the player decides on a legal move. Since it is not possible to take illegal actions, this field was added to ensure that no processing time is spent by the respective heuristic models evaluating possible moves that will simply turn out to be unpermitted. All possible moves will be included as a single string separated by a "—" between each value.

The fifth field is Goal_State. The agent will also include its current goal state to reach in the request. This will allow the heuristic model processes to evaluate the possible moves and evaluate the desirability of each with respect to the goal state.

The last field is Respond_By. This field allows the agent to set a firm deadline for all processes to respond by. How far the agent sets the deadline in the future can be varied from one run to the next to find differing balances between how fast the agent makes a decision and how well thought out the decision taken is. Any process that has failed to respond within the designated time frame will not have the opportunity to influence the agent's decision for the current input opportunity.

Agent_Response - The purpose of this table is to respond to the requests which have been sent out by the agent. Each heuristic process will be monitoring for requests and will post its response in this table, which the agent will check at the designated deadline.

This table has a Request_Uniq field, which is a foreign key to the Agent_Request table. It also has a field called Goal_Prob which contains the probability that a

particular option will result in the goal state. It is encoded as numeric percentages with two decimal precision with each value separated by a "—" with the percentages in the same sequence the options were in the request. The last field is Confidence_Factor. The purpose of this field is to inform the agent of the degree of confidence it should place in the predicted probability of reaching the goal.

Transition - The purpose of this table is to hold the complete model of all the transitions that a particular heuristic has constructed. The algorithm by which the heuristic populates this table is dependent on the particulars of the pattern type(s) that the heuristic is designed to search for. It consists of nine fields. The first field is the uniq field and its purpose has already been explained. The second field is the Game_Type field. Its purpose is to identify to which game type the current row entry belongs.

The third field is the Heuristic_Type field. This field is used for the purpose of keeping any model built for the specified game type separated by the heuristic that built the model. The values in this case are simply the name of the heuristic, such as "Observed State." Not all heuristics will actually build a model that is stored in the database. Some heuristics are simply designed to help fill in gaps in the models already built by other heuristics.

The fourth field is the Cur_State field. This field stores the current state that the heuristic building the model believes the state to be. In the case of the observed state heuristic, this would simply be the currently observed output of the problem. For the purposes of these games, that is the equivalent of the current position of the board. For other heuristics, while the observed output is certainly important, the current

state may consist of something beyond the directly observable information.

The fifth field is Input. This field stores the input that was given at the current state for this row entry. The sixth field is Next_State. This is the state that the current FA model is thought to have transitioned to after receiving the input. The analysis of how that is determined is once again dependent on how the heuristic building this FA determines what the state is.

The seventh field is Stat_Count. This field holds the count for how many times this particular transition has occurred. This is especially important in the case where the only accurate model that can be built for the problem is that of a PFA. Since a PFA can not be perfectly predicted as to what the result will be on a given input, counting the number of times a particular transition has been chosen in relation to another can then be used to evaluate which transition is the most likely outcome.

The eighth field is End_States. This field is used to aid in creating a tractable approach to deciding what action to take. It effectively maintains a list of the ultimate end state(s) that have been reached when the game ended based on the transition that the current row represents.

The ninth and last field of the Transition table is End_State_Count. This field maintains the statistics of how many times the end state(s) that a particular transition has led to occurred. This enables a very fast comparison based on the statistics to determine what action to take. Once again, the purpose of this field is to help maintain at least one decision criteria that the agent can use that will always be tractable.

Game_Type_History - This table is used to maintain a master list of every game that is played using the game engine. It has four fields. The first is Id, and is the

unique identifier for the game. The next is Type, which simply stores the type of game in the manner that has been established earlier. The third field is Started, and is used to record when the game began. The last is Ended, and it records when the game ended.

Game_History - The purpose of this table is to maintain a history of every move made in every game that is played in the game engine. The history will be recorded whether the players are human, FABLE agents, or random.

The uniq field has the same purpose as was already described. Game_Id is a foreign key that links back to the Game_Type_History table on the Id field. The field Step is to maintain the order in which each of the observed states in the game's history occurred. Cur_State maintains a history of the observed state at the time of this step. Last is Input, and it is used to maintain a history of what input was received from the current state.

Game_Processed - The purpose of this table is to keep a record of which games have been processed for each heuristic. This enables heuristics to efficiently build transition tables without needing to repeat work already done from the processing of games which have already been incorporated into the transition table for that heuristic. This table has 3 fields.

The first is the uniq field which is already understood. The second is the Processed_Games field which stores the ID from the Game_Type_History table corresponding to the game that has already been processed. The last field is the Heuristic_Type field which specifies for which heuristic the given ID has already been processed. A heuristic makes the appropriate entry in this table after processing the

information for a game from the Game_History table.

Now that the entire the database schema has been given and explained, we can see that all the information necessary to re-create any existing game will be stored. This will in turn allow any heuristic to peruse the complete history of every game for the purposes of building and testing FA models of the problem. A table structure is also present for the agent to make a request that allows its decision making to be distributed among separate processes that will each evaluate the position based on a particular heuristic and then send a response. With this we can now look at the design of the heuristic algorithms.

3.3.3 Heuristic Algorithm Design

The design for each heuristic that will be implemented for the current version of FABLE are the observed state heuristic, the nearest neighbor heuristic, the locality heuristic, and the rotationally invariant heuristic. Each heuristic will be implemented as a stand-alone executable which can connect to the FABLE database. This structure allows the processing to be distributed among multiple cores, or even among multiple computers on the same network. The UI for each of the heuristics will consist of a simple interface to turn the processing for the individual heuristic on or off.

The observed state heuristic works under the assumption that the state information that can currently be observed is in fact the total state. The basic algorithm for constructing a FA using this heuristic will be to query to the database to see if there are any completed games that have not yet been processed. Once a game that needs to be processed is found, the entire history of the game will be added to the Transition

table in the database. If the transition has been seen before, then the statistics for that transition will be updated, otherwise a new transition will be created showing the observed state before the transition, what the input was, and the observed state that followed that input.

The nearest neighbor heuristic will analyze any existing FA model generated by any of the other heuristics. The method of analysis is to look at how closely related the start and end states for transitions are for those states that are neighbors of each other. As an example, let us suppose we have the situation in a game of minesweeper as depicted in Figure 3.4. As stored in the database, the current observed state would be represented as "|1||1|2||2|2||2|." The distance between neighbors is measured by evaluating how many squares in the grid differ. Thus "|1||1|2|||2||2|" would be a neighbor that is only a distance of 1 away from the current observed state, while "||||2|||2|||" would be a distance of 3 away from the current observed state. FABLE will not actually build and store a complete nearest neighbor FA for each FA that has been generated by other heuristics. Instead, a small nearest neighbor FA will be constructed in memory on demand that attempts to fill in the gaps in a current model using neighbors that are no more than a distance of 1 away from the current state.

The locality heuristic, similar to the observed state heuristic, will construct a FA by analyzing the game history. The difference is that whereas the observed state heuristic builds a FA using the entire state for its transitions, the locality heuristic will only look at a sub-set of the state at any given time. The degree of specificity of the locality is determined by how large of a chunk of the total state is being

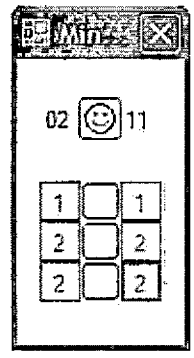

Fig. 3.4: User Interface Depiction of Minesweeper State "|1||1|2|||2||2|".

evaluated. For FABLE, the specificity will be to build a locality based FA for states that look at only 1 spot on the grid, a 2 * 2 square of the grid, and 3 * 3 square of the grid and a 4 * 4 square of the grid. It is possible to create localities that are not square, including localities that are not even comprised of contiguous parts of the observed state, but FABLE will only look at square localities up to a size of 4 for the current implementation. The method of construction is the same as that for the observed state except that the observed state information in the game history will have the information that is not part of the current locality removed. This heuristic will essentially be looking for patterns that are predictable based on a subset of the data. The usefulness of this approach is that by "throwing away" the additional information, the potential complexity of the FA is greatly reduced. This should greatly increase the speed of learning for those problems where locality is a principle that applies to the problem.

The last heuristic is the rotationally invariant heuristic. The purpose of this heuristic is to look for patterns where all that matters is the relative position of the board

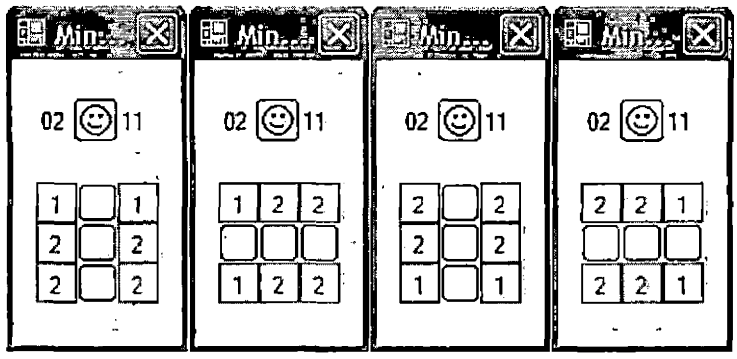

Fig. 3.5: User Interface Depiction of Minesweeper State "|1||1|2|||2||2|" and its 3 Rotations.

rather than the absolute position of the board. The rotationally invariant heuristic is similar to the nearest neighbor in that this heuristic will not build a FA that is stored in the database on its own. Instead, it will build a model in memory on demand for the current state as reported to it by the part of the agent residing in the main game engine. The method the rotationally invariant heuristic uses to construct its model is by taking a current state for a model and rotating the values in the grid about its axis. The current design for the rotationally invariant heuristic will build a FA in memory based on the original orientation, a 90 degree orientation, a 180 degree orientation and a 270 degree orientation. An example of these orientations is found in Figure 3.5 showing a visual representation of the minesweeper state"|1||1|2||2|2||2|" in its original, 90, 180, and 270 degree orientations respectively. Once again, this is used primarily to reduce the number of states that needs to be explored and will only be helpful for those problems where the orientation will not result in different outcomes.

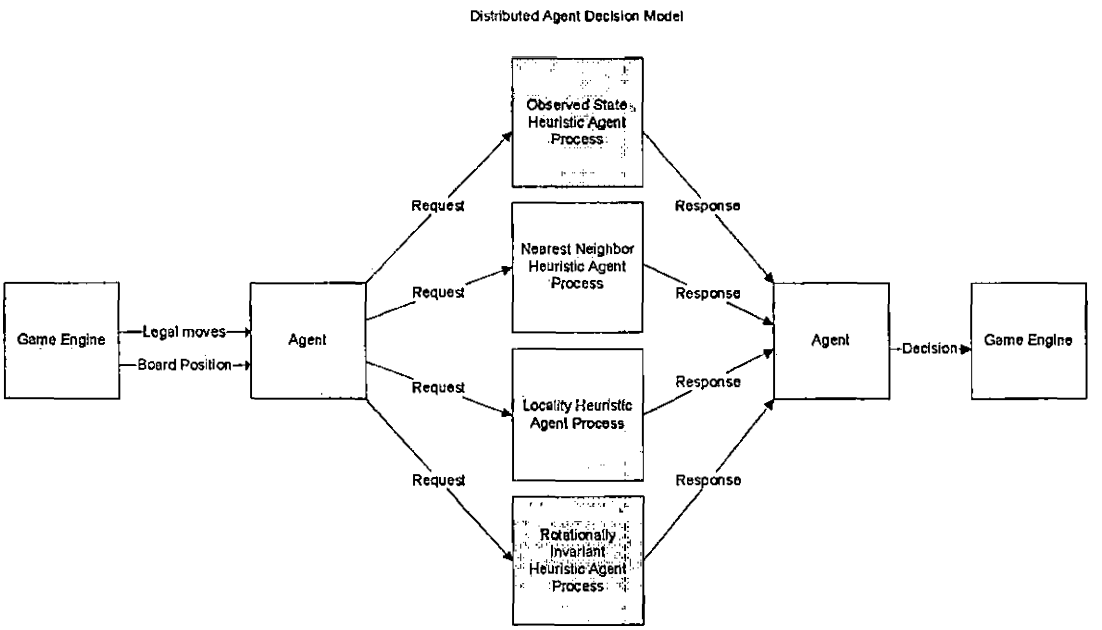

Fig. 3.6: Basic Agent Algorithm Workflow.

3.3.4 Agent Algorithm Design

The agent has been designed to be distributed to 5 different processes for FABLE. The agent will consist of one part residing inside the game engine. This part of the agent is responsible for ascertaining the current observed state as reported by the game engine, the current legal moves as reported by the game engine, and for supplying the game engine with a selected move or input. The agent will select a move by evaluating the relative "goodness" of each move as reported to it by each of its distributed heuristic based processes. Each heuristic executable will also contain a portion of the distributed agent and is responsible for evaluating how "good" each option is based on the FA model for that heuristic. The basic workflow can be found in figure 3.6.

We can see from the figure that the individual heuristic agent processes all run concurrently. Due to the requirement to maintain tractability, the evaluation in each of the heuristic processes does not actually attempt to traverse the tree of possible next states. Instead, a statistical count is kept for each state as to what end state ultimately resulted from the transition and is used for making a quick evaluation as to the probability that the selected transition will lead to the desired goal state or at least avoid undesired end states.

The basic method for evaluating any potential move by the agent processes will be to look at the total number of times the move (transition) has led to the desired state and compare that with the total number of times the move has led to undesired states. A probability for reaching the desired goal state is then computed. In addition, a confidence factor will be computed at the level of each move. The method employed by FABLE will be to multiply the probability by the confidence factor for the overall "goodness" of choosing a move. Once all possible moves have been evaluated, the agent process will compare all computed values for a move and select the one with the greatest absolute value for its response.

The portion of the agent running inside the game engine follows the same basic algorithm as found in the agent processes. Once it receives the response from the heuristic agent processes, it evaluates the responses for each move and selects the response with the greatest absolute value to be the value it will use for evaluating that move. Once the combined list of absolute values has been created, the agent will sort the list such that the best moves are at the top and the worst moves are at the bottom. For the final determination of what to do, the agent will loop through

the entire list. For each iteration, a random number will be generated and compared to the current item in the list. If the random number is less than the value of the current item, then the agent selects the move and sends its response to the game engine. Otherwise, it continues to the next item in the list. In the event that the agent loops through the entire loop and an item has still not been selected, it will loop through the entire repeatedly until a selection is made. In the highly unlikely event that the loop has been repeated 10 times, the agent will give up and report to the game engine to simply select a move at random.

3.4 System Implementation

System implementation followed the designs. The database structure was created on MS SQL Server. The heuristic processes were implemented according to the designs given above and the distributed portions of the agent were implemented within them as well. For those heuristics that actually create multiple models or predictions, such as the locality heuristic, the coding made use of multiple threads so that each model or prediction could be run concurrently. The agent process in the main game engine was implemented according to the above designs and functions as expected. The game engine was implemented such that the rules for each game as they were explained at the beginning of this chapter are followed.

3.5 Evaluation of the System

Now that the complete design of FABLE is established, the next step to consider is the methods by which it will be tested. Since there are 3 games to be tested, each

with different rules and complexities, the expected results and means of testing will differ slightly from one to the next. The specifics in each case will actually be covered in depth in Chapter 4, but there are some aspects of how the testing will be done that can be generalized.

In order to establish whether or not FABLE is learning, the first benchmark in all cases will be to establish what the expected results of an agent making choices in a purely random fashion are. The expected results of making choices may be found by ab initio methods using probability theory or empirically by simply creating a random agent collecting data on what results the random agent actually generates. In some cases, the expected results of random choices, or random play, will be determined using both methods. In this case the theoretical results from probability theory should match the actual observed results.

Once the results of random play have been established, the next benchmark that will be established is the expected results for an agent that makes only perfect decisions. Perfect play is defined as making the decisions that maximize the likelihood of achieving a goal state while minimizing the likelihood of undesirable states. An example is that a game that can end in a win, a loss, or a neutral state such as a draw. A perfect agent will be able to force a win whenever forcing a win is possible from the current position. In the event that a win can not be forced, a perfect agent will make choices that do not allow a loss if possible. If a win can not be forced, and a loss can not be avoided, a perfect agent will at least make choices that delay the loss from occurring for as long as possible.

In theory, an agent that always makes perfect choices for any task that is simple

enough to model with an FA exists. In practice, it can be extremely difficult to determine what perfect play actually is for a specific task. For each game, perfect play will be evaluated to the extent that can be reasonably done. In the event that perfect play can not be determined, it will not invalidate the results of testing for the task. It should still be possible to measure improvement compared to making choices at random. Rather, the concept of perfect play is only introduced in an attempt to find the upper limit of what can be accomplished by even the most intelligent agent possible. Having this second benchmark when possible allows FABLE to be evaluated not just in the since of whether it does better than random, but in terms of how close it gets to learning the task perfectly.

The data for each of the 3 games will be gathered by using different combinations of random and FABLE agents. For each combination a data run will be conducted that will use the same combination of random and FABLE agents for a specified number of games in a row. The outcomes of those games will then be analyzed using various methods including the use of averages, standard deviations, regression, and making observations based on the data after it has been graphed. The specifics of which techniques are used in what manner will be given for each game in more detail in Chapter 4. This is done so that the specifics of each game can be evaluated, and the results can be given directly following that evaluation.

3.6 Summary

In this chapter, a list of requirements, the design, and the implementation of the FABLE system were given. The basic structure of FABLE is seen to consist of an

agent that observes and interacts with the problem. In this case that is the game engine. The agent then distributes the work of deciding what to do among various heuristic based processes and makes a decision based on the results each process reports back. Each heuristic process enables the portion of the agent residing in it to make that evaluation by creating a FA model of the problem that is either stored in the database or generated in memory on demand. Each heuristic is designed to look for specific types of patterns in order attempt to model the problem in a tractable time.

4. RESULTS

4.1 Introduction

The purpose of this chapter is to show how FABLE was tested to determine if it does indeed show the learning abilities that it is hypothesized to have. The results will be taken for each of the three games in the game engine starting with minesweeper, moving on to tic-tac-toe, and ending with checkers. For each game, the complexity of the game will be considered. A discussion of how to derive some properties of the game ab initio will follow. Once the game theory is established, the actual results for the specific game will be given, and those results will be compared against the type of performance that was expected. A general summary of the results will then be given.

4.2 Minesweeper

4.2.1 Game Theory

Game Tree Size

The first property of minesweeper that I will derive is how to measure the size of the game tree. In "Learning minesweeper with multirelational learning" Lourdes Castillo calculates the size of the game tree for a specific instance of minesweeper on a 8x8 board with 10 mines [7]. However, his calculation was not written in a generalized

form. In addition, his calculation is only for the game tree size for an individual game where the mines have already been arranged. His calculation further requires that the configuration of mines be such that there are no cells in the grid that are not adjacent to a mine. My derivation will establish an upper and lower bound for the game tree size and will be generalized to grids of various sizes. My formula will also take into consideration the various possible configurations that may be present at the start of the game. An important note is that my formula does rely on a fixed 20% mine density.

To accomplish this derivation, let us first understand that minesweeper is effectively a game of process of elimination. At the start, all of the cells in the grid are available to be clicked. As a cell is clicked with each subsequent turn, the number of possible cells left to be clicked will be reduced by at least one. The only two possible end states for minesweeper are to win or lose. This implies that the game tree size can be found by adding all of the possible ways to win to all of the ways to lose.

Let us define a as the number of cells in the grid and b as the number of mines to be placed. For any game, the first thing that must be determined is where the mines are located within the grid. The total number of ways to do this is simply $\binom{a}{b}$. Determining the exact size of the game tree for a given set of values for a and b is intractable for all but the smallest of grids. However, the size of the game tree can be given a lower and upper bound to at least get an approximation of the size of the game tree.

For the sake of simplicity, the lower bound will be found by simply stating that for every possible mine configuration, there is at least one way to lose and at least one

way to win. In reality, almost all mine configurations on almost all grids will have a much larger number of possibilities, but none will have less. This means that since we already know the number of mine configurations for a grid to be $\binom{a}{b}$, that since there are at least 2 possible outcomes for each these, a valid lower bound the size of the game tree is $2\binom{a}{b}$. This result is sufficient to show that the game tree size will grow at a factorial rate on the lower bound.

To find the upper bound, we can simply ignore the possibility of any cell causing its neighboring cells to be clicked automatically. To win under this condition requires that the first click consist of one of the non-mine cells, of which there are $(a - b)$ possibilities. Each subsequent click must click one of the remaining non-mine cells until only mine cells remain. Since automatic clicking is being ignored, the result will be the same for all mine configurations, resulting in equation (4.1). Finding the upper bound of the ways to lose is done by counting b possible ways to lose for every way of surviving to the current turn. Surviving to the current turn is the same as following the winning sequence up to the last turn. For example, if a is 9 and b is 2, then on the first turn there are 2 ways to lose. To lose on the second turn, an agent must click one of the 7 non-mine cells on the first click, and then click one of the 2 mines on the second click. This becomes $2 * 7$ ways to lose on the second click. For the third click it becomes $2 * 7 * 6$. The process continues until the second to the last non-mine cell is clicked. The upper bound of the ways to lose is then the summation of the ways to lose on each click. Generalizing this method in terms of a and b, the upper bound on the ways to lose is found in (4.2).

$$\text{Upper bound of ways to win} = \binom{a}{b}(a-b)!$$
$$= \frac{a!}{b!} \tag{4.1}$$
$$\text{Upper bound of ways to lose} = \binom{a}{b}\sum_{i=1}^{(a-b)}\frac{b(a-b)!}{i!}$$
$$= \binom{a}{b}b(a-b)!\sum_{i=1}^{(a-b)}\frac{1}{i!}$$
$$= \frac{a!}{(b-1)!}\sum_{i=1}^{(a-b)}\frac{1}{i!} \tag{4.2}$$

Since the density of the mines will be held constant at 20%, b is actually just $\lfloor a/5 \rfloor$. Considering that $\lim_{a\to\infty}\lfloor a/5 \rfloor = \infty$. This can be used to put an upper-bound on the summation found in equation (4.2).

$$\sum_{i=1}^{(a-b)}\frac{1}{i!} <= \sum_{i=0}^{\infty}(\frac{1}{i!}) - 1 = e - 1 \tag{4.3}$$

Replacing the summation in equation (4.2) with the result from equation (4.3) yields the final result for the upper bound of the ways to lose:

$$\text{Upper bound of ways to lose} = \frac{a!(e-1)}{(b-1)!} \tag{4.4}$$

The upper bound for the game tree size of minesweeper can be found by adding the upper limit of ways to win from equation (4.1) to the upper limit of ways to lose in equation (4.4). Remembering our lower bound to be $2\binom{a}{b}$, we can now show the final result for the game tree size in equation (4.5).

$$2\binom{a}{b} <= \text{Game Tree Size} <= \frac{a!}{b!} + \frac{a!(e-1)}{(b-1)!} \tag{4.5}$$

Probability of Winning by Chance

Now that the game tree size of minesweeper has been bounded, the next property of the game to determine is the likelihood of winning by chance. Castillo calculated a value for this on a 8x8 board with 10 mines present and no cells without a mine adjacent, but did not generalize the formula. In order to find a more accurate measure of the actual probability of winning at random, I will describe an algorithm which can find the actual probability of winning given the different possible mine configurations as well as the possibility of cells without mines adjacent to them.

To calculate the odds of winning a game of minesweeper by chance, the basic method is to start by determining all of the configurations that the mines can be arranged in at the start of the game. While evaluating the game tree size, this was already determined to be $\binom{a}{b}$. Let us define c as this value:

$$c = \binom{a}{b} \tag{4.6}$$

Next, we can define a function $MC(x)$ which receives an integer value and returns the mine configuration corresponding to that value. Let $MCProb(x)$ be a function where x is a mine configuration and the return value is the probability of winning a game played on that configuration assuming random selection. In this case, the total probability of winning becomes the expression found below:

$$\sum_{i=1}^{c} \frac{MCProb(MC(i))}{c} \tag{4.7}$$

Unfortunately, expression (4.7) is useless unless we know how to compute the value of $MCProb(x)$. Evaluating this function is not simple due to the possibility of cells in the grid which are not adjacent to any mines. Clicking one of these cells triggers the automatic clicking of their neighbor cells. If not for this automation, the function could be evaluated quite easily. This would be done by letting d be the number of un-clicked cells remaining in the grid. In this case, the probability of winning by clicking at random would be found in expression (4.8).

$$\frac{b!(d-b)!}{d!} \tag{4.8}$$

Due to the automation, for any mine configuration in which automation will be part of the winning sequence, additional work must be done before expression (4.8) can be applied. Since a rigorous method of evaluating $MCProb(x)$ appears to be complex, let us first look at how to generate a good approximation. The first step is to separate all non-mine cells into three groups. The first is comprised of the cells which must be clicked under all possible paths that lead to a win. The second is comprised of the cells which can be automatically clicked if a neighboring cell triggers automation, but are not able to trigger automation themselves. The last is comprised of the cells that trigger automation.

With these groups, the function $MCProb(x)$ can be defined recursively as a function that finds the probability of winning if a cell in the first group is clicked by calling

itself with the number of cells in the first group reduced by one. It then proceeds to evaluate the probability if the click was in the second group. Last, it evaluates the probability in the event that a cell in the third group is clicked. When no cells in the second or third groups are present, it evaluates to the value given by (4.8). In each case, when the recursive call returns, the returned value is multiplied by the number of cells in the group and divided by the total number of cells that can be clicked from the current position.

A exact method of finding the probability of winning actually requires separating the cells into the same groups as the approximation method, but then requires that the second and third groups be further broken up into sub-groups. The method of generating the sub-groups requires that the cells in the third group be broken up into separate cascades. That is, the automation that occurs when a cell that is not adjacent to any mines is clicked can result in the clicking of a neighbor that also is not adjacent to any mines. This automatically clicked neighbor then clicks its neighbors. This process will continue until the supply of adjacent cells that are not adjacent to any mines is exhausted. All cells in the third group that trigger the clicking of each other are put into the same cascade sub-group. The defining property of a cascade is that it does not matter which of the cells in the cascade is clicked, the end result following the click will be the same as clicking any other member of the same cascade. The cells in the second group are then separated by which cascade they are clicked by. A special case to consider is when a cell in the second group is actually a member of two separate cascade sub-groups.

The method of evaluating $MCProb(x)$ is now modified such that each sub-group

is treated separately in the evaluation. In the event that a cell in the second group is a member of more than one cascade sub-group, it must be treated as a click which reduces the number of cells available in all of the cascade sub-groups which it is a part of. This algorithm results in a perfectly accurate result, but it is also intractable except for small grids.

This method of determining the probability was encapsulated within a small program and was run on a 3x3 grid with 2 mines. The result was that the chance of winning by playing at random for this grid was determined to be 15%. Finding the value for a 4x4 was attempted, but was taking too long to justify continuing execution when available resources were needed to run FABLE. Due to this, the probability of winning at random for grids larger than a 3x3 have been left to be determined empirically.

Probability of Winning by Perfect Play

The problem of determining perfect play for a given minesweeper position is shown to be equivalent to the "Minesweeper Consistency Problem" by Richard Kaye in "Minesweeper is NP-complete!" [14] Kaye's proof effectively shows that the "Minesweeper Consistency Problem" is the equivalent of the well known SAT problem. This proof does not give an algorithm for finding the probability of winning given perfect play.

A method for determining the probability of winning by perfect play would be to determine which cell to click corresponds to perfect play. Since determining this is of the equivalent complexity as the "Minesweeper Consistency Problem," finding the probability is of at least the same complexity as solving a NP-Complete problem.

This means that finding the probability of winning by perfect play is intractable for all but the smallest of grids. Despite this, I will describe an algorithm that I designed which can be used to find the probability for small grids.

Determining the probability of winning by perfect play is actually a simpler problem than determining the probability of winning by chance. This is due to the fact that any choices that are obviously less than optimal can be ignored. The simplest example of this is that if the currently known information is enough to prove that a particular cell contains a mine, then the possibility of clicking that cell no longer needs to be considered. There are also situations where the available information can prove that a particular cell is not a mine. In this case there is no risk in clicking the cell, and the proper action is therefore to click it.

The difficulty in determining the probability of winning by perfect play comes from when there is not an obvious choice. This occurs when there is insufficient data to prove that any of the cells to be clicked is not a mine. Depending on the information that is available, it may be possible to assign higher and lower probabilities of a cell containing a mine, in which case the general strategy would be to click cells with lower probabilities of being a mine first, but depending on the specifics of the situation, there are exceptions to the general strategy.

The basic method of determining perfect play is recursive in nature. It involves looking at the current information and finding all of the possible mine configurations that are still possible. For any individual cell, the probability that it is a mine can be found by finding the total number of possible mine configurations where it is a mine and dividing it by the total of all possible mine configurations. In addition to

the probability that a cell is a mine, the probability of how many mines the cell is adjacent to can be found in a similar manner.

Clicking a cell can be evaluated by making a recursive call to the function where the cell has been clicked for each of the possible outcomes remaining for the cell. That is, if a cell might have 1 adjacent mine or 2 adjacent mines, then a recursive call is made where clicking the cell resulted in a 1 and another call is made where clicking it resulted in a 2. The return value of each outcome is multiplied by the probability of that outcome resulting from the click. These probabilities are then summed together. Finding the perfect cell to click is done by evaluating each available cell to click. The cell with the highest probability of winning is the perfect choice. The probability of winning by clicking this cell is the return value of the recursive function.

The efficiency of this algorithm can be improved slightly. This is done by ignoring the possibility of any click which obviously can not have a higher probability of winning than the current leader. Despite this, the algorithm is still intractable for all but the smallest of grids.

Using this algorithm the probability of winning on a 3x3 grid with 2 mines where each click was the perfect choice was calculated to be 66.67%. No attempt was made to find the value for a larger grid size.

4.2.2 Data Results

The data for minesweeper was generated on 3x3, 4x4, and 5x5 size grids. The 20% mine density used corresponds to 2 mines for the 3x3, 3 mines for the 4x4, and 5 mines for the 5x5. In order to establish a baseline for performance, a run of 10000

Tab. 4.1: Wins per 100 Games Played Randomly on Minesweeper.

Grid Size	3x3	4x4	5x5
Average	12.5	2.5	0.8
Maximum	21	7	3
Minimum	4	0	0
Standard Deviation	3.32	1.58	.88

games was played at random for each grid size. To establish that FABLE is capable of learning to play minesweeper, the method of measuring what it has learned will be by determining that FABLE starts out with a win/lose ratio on par with making selections at random. Over time, FABLE is expected to improve the win/lose ratio. FABLE was used to play 10000 games of each grid size as well, with the expectation that its performance will improve over the course of those 10000 games.

To empirically determine the probability of winning by chance, 10000 games were played and then broken down into groups of 100 games each. This yields 100 groups of 100 games. For each set of 100, a tally was made of how many games were won versus how many games were lost. The averages, minimums, maximums, and standard deviations of these data runs are found in table 4.1. An individual chart comprising all the data points has also been created for each of the grid sizes. These charts can be found in figures 4.1, 4.2, and 4.3.

These empirical results seem to match up very well to theory. For the 3x3 grid, the predicted probability of winning by chance was 15%. The empirical probability was 12.5% with a standard deviation of 3.32%. This means that the predicted value is

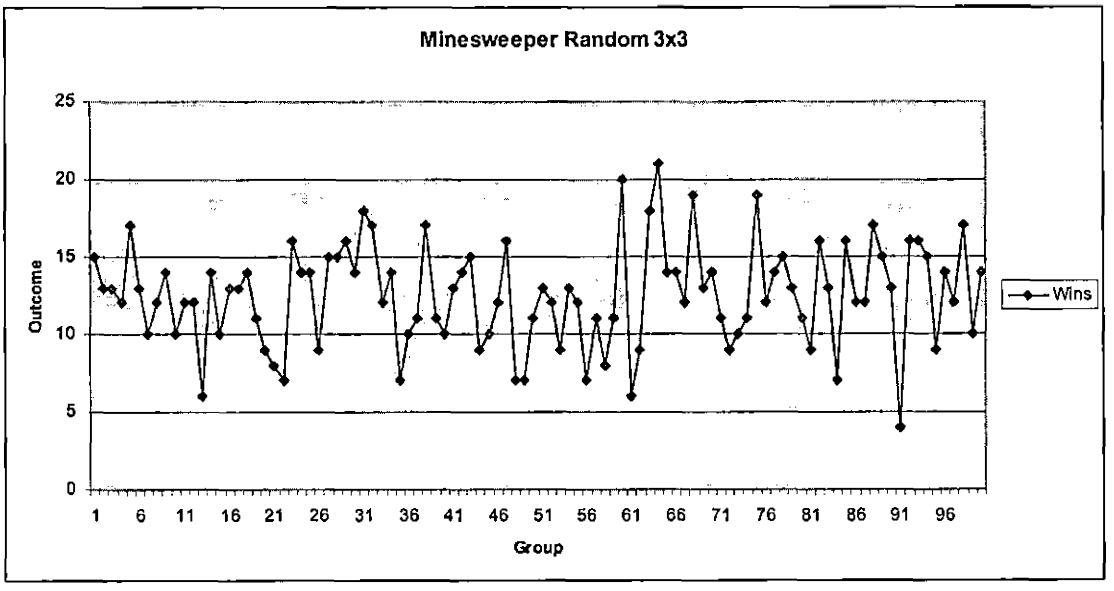

Fig. 4.1: Wins per 100 Games Played Randomly on Minesweeper 3x3.

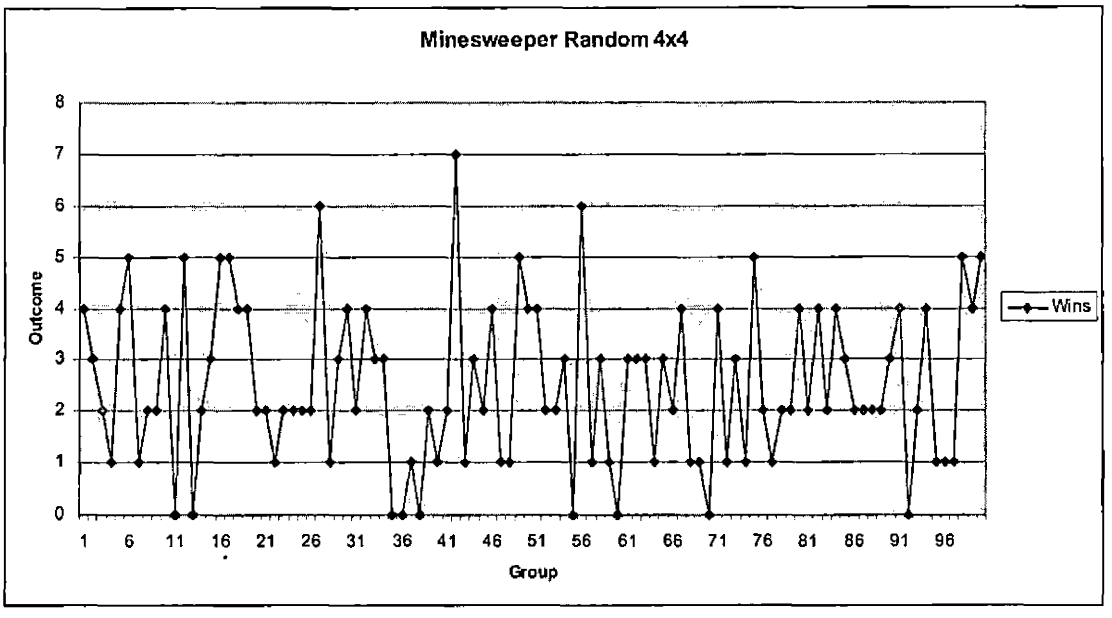

Fig. 4.2: Wins per 100 Games Played Randomly on Minesweeper 4x4.

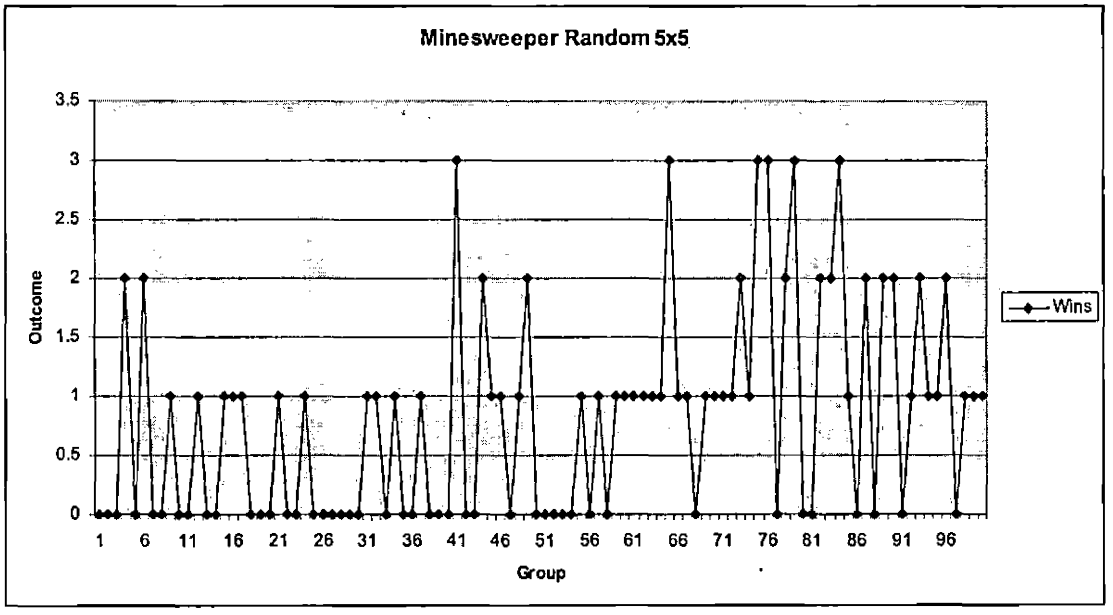

Fig. 4.3: Wins per 100 Games Played Randomly on Minesweeper 5x5.

only 0.75 standard deviations away from the observed win rate. This establishes that theory and observed values are in agreement with each other. Given the reliability of the result for the 3x3 grid, it is probably safe to accept the empirical values for the 4x4 and 5x5 grids as being accurate for what the theoretical values would be if they were calculated.

Now that the baseline performance for playing at random has been established, we can turn our attention to how to measure improvement. The expected learning behavior for FABLE is to learn in discrete steps. This is due to the fact that the information from a game is not processed until the game has terminated. These steps may be of various sizes depending on how much information a newly processed game adds to the transition table that was not there before. A game where the first cell clicked was a mine will add little to the transition table. A game that lasted for 12 clicks adds much more to the table, and it will likely improve future performance to

Tab. 4.2: Wins per 100 Games Played by FABLE on Minesweeper.

Grid Size	3x3	4x4	5x5
a	51.96	70.1	35.48
b	1.2	1	1.04
c	0.12	0.04	0.01
Horizontal Asymptote	62.35	72.9	36.9
Standard Deviation	5.88	4.73	2.92
Correlation Coefficient	0.97	0.97	.78

a larger degree.

Despite the step-wise nature in which FABLE learns, as well as the inherent variability in outcomes due to the random nature of minesweeper, it should still be possible to find a continuous function that at least is a good approximation of the learning curve. As was done with the random data, the 10000 games will be grouped into sets of 100. Looking at the results 100 games at a time will help to smooth, the inherently step-wise learning curve. Several different function forms were analyzed for the data to find a good fit when regression is applied. The selected function form is $y = a(b - e^{-cx})$. This is due to the fact that when applied to the data for the 3x3, 4x4, and 5x5 grid sizes, it shows high correlation coefficients coupled with relatively small standard deviations. The results of this analysis are found in table 4.2. In addition, charts with the data points plotted on them can be found in figures 4.4, 4.5, and 4.6.

The fact that $y = a(b - e^{-cx})$ is a good approximation of the actual learning curve

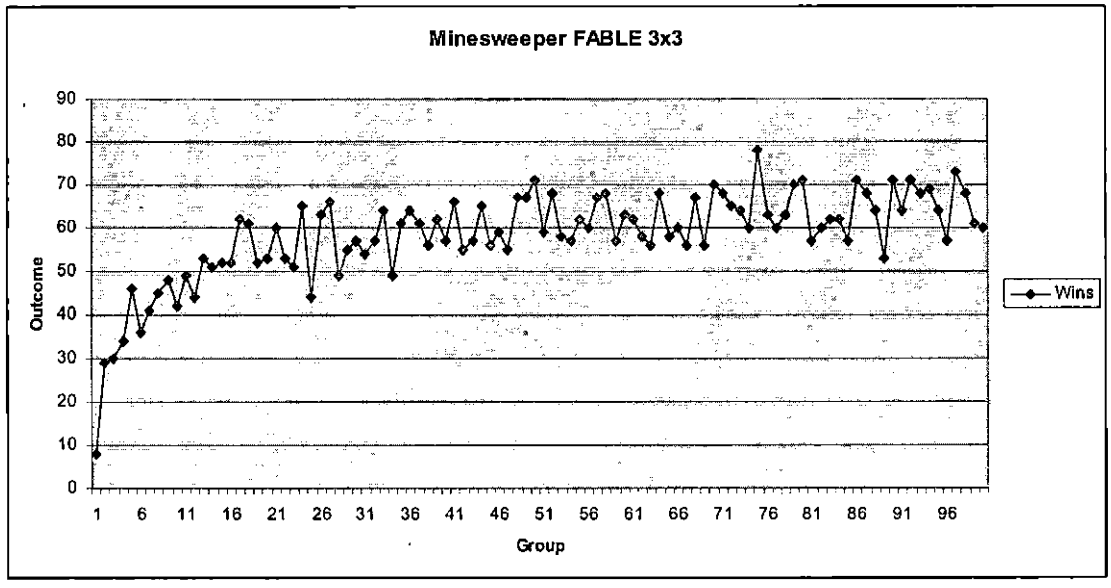

Fig. 4.4: Wins per 100 Games Played by FABLE on Minesweeper 3x3.

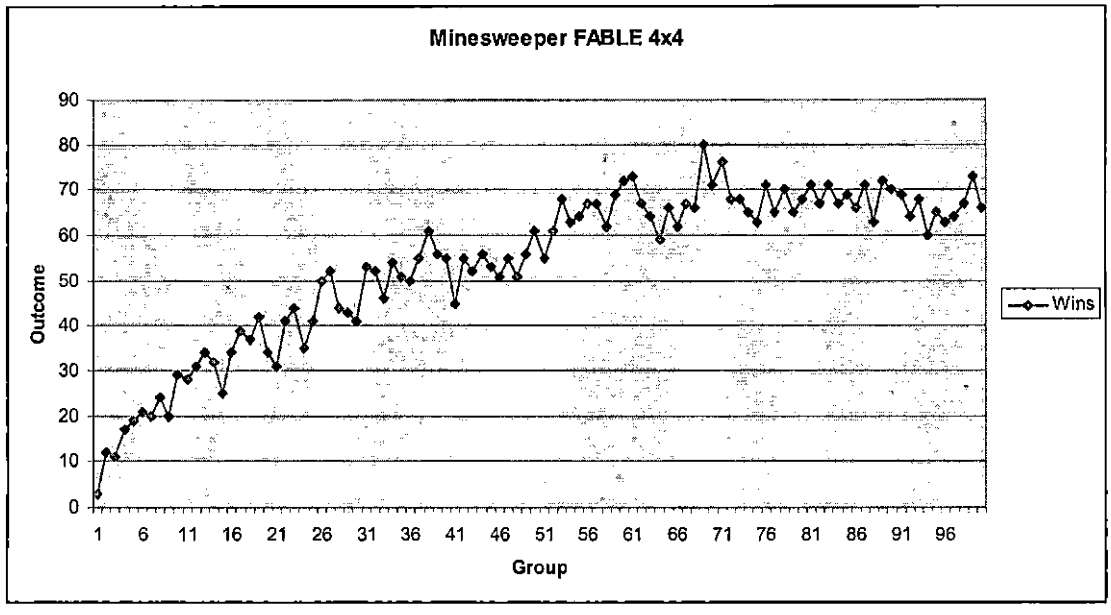

Fig. 4.5: Wins per 100 Games Played by FABLE on Minesweeper 4x4.

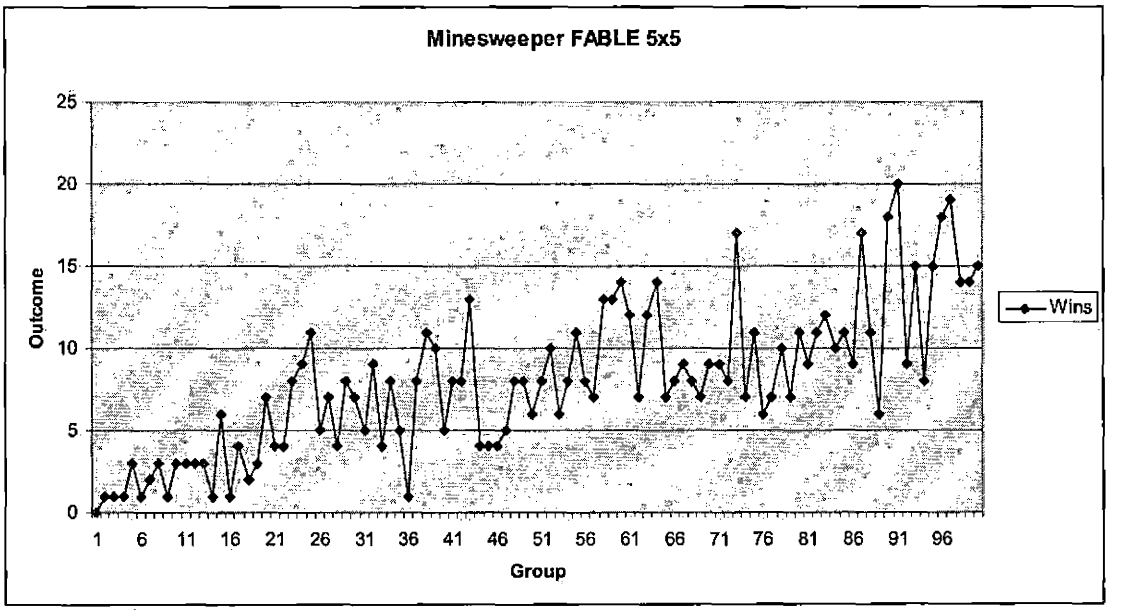

Fig. 4.6: Wins per 100 Games Played by FABLE on Minesweeper 5x5.

allows us to describe the characteristics of FABLE's performance as it pertains to minesweeper. The first characteristic is that there is a horizontal asymptote which represents a level of performance that FABLE will not be able to improve upon. This is found by simply multiplying a by b, since $\lim_{x\to\infty} e^{-cx} = 0$. That is how the horizontal asymptotes in table 4.2 were found.

It is interesting to note that the horizontal asymptote for the 3x3 grid is 62.35% with a standard deviation of 5.88%. The predicted value for playing perfectly on the 3x3 grid is 66.67%, which is only 0.73 standard deviations away from the horizontal asymptote. A glance at figure 4.4 shows that the actual performance was already very close to this value as well. This is not sufficient to prove that FABLE learned to play minesweeper on a 3x3 size grid perfectly, but it does demonstrate that it at least learns to play at a level that is very close to perfect play.

The only way to prove that it learned to play perfectly would be to show that based

on the information in the transition table, FABLE will always choose the move that perfect play would choose. To do this would be tedious, so instead it will be argued that the level of performance is statistically indistinguishable from perfect play. This means that while perfect play has not been proven to be achieved by FABLE on the 3x3 grid, the statistics do not prove that FABLE's performance is anything less than perfect either.

As to the question of whether or not FABLE learns to play minesweeper at level that is indistinguishable from perfect play for the 4x4 and 5x5 grids, I can only offer reasoned arguments. This is due to the intractability of computing the actual probabilities of winning by perfect play on larger grids.

For the 4x4, the horizontal asymptote seems to be a reasonable value for what is achievable by perfect play. Considering that the odds of losing are $\frac{3}{16}$ on the first click, it is impossible for perfect play to result in a probability of winning that is greater than 81%. Even if perfect play is a full standard deviation less than the projected horizontal asymptote of 70.1%, such a high probability of winning implies that for the majority of possible mine configurations, the game can be won with almost 100 probability after the first click. I can attest that this is often the case from personal experience. The reason this is achievable is due to the still relatively small size of the grid. For instance, if the first click reveals that a cell has 3 mines directly adjacent to it, then all of the mines in the entire grid are adjacent to that cell. Therefore, all cells that are not directly adjacent to that cell throughout the rest of the grid are safe to click. After clicking those cells, there is usually enough information about the grid to win without taking any chances. There are several other examples of this type of

reasoning that allow a high win probability.

For the 5x5, lines of reasoning that appeal to limiting the possible mine configurations based on the total number of mines left in the grid become much harder to use. Using the same example from the 4x4, if the first cell clicked reveals a 3, there are still two mines that are unaccounted for. This means that it is not safe to assume all cells that are not directly adjacent to the first cell clicked are safe to click. In fact, even with perfect play, the odds of winning should decrease as the size of the grid increases simply due to the fact that the larger the grid, the greater the chance that there will be at least one part of the grid where the mines will be arranged in such a fashion that there is no way to ascertain where they are without taking a chance at least once. On average, as the grid gets larger, so should the average number of times such configurations arise within the grid in at least one spot. The horizontal asymptote of 35% for the 5x5 grid is therefore at least a plausible value for the odds of winning by perfect play.

Another characteristic that the analysis shows is that as the size of the grid increases, the number of games that FABLE must play before performance saturates increases as well. This is reflected in the decrease of the value of c as the size of the grid increases. As the value of c decreases, the performance growth curve becomes shallower since e^{-cx} approaches 0 at a slower rate. It is possible that c is decreasing because with the much larger number of possible games, it simply requires a larger sample of games to acquire sufficient information before performance can be improved.

Another explanation for why c is decreasing as the size of the grid increases can

be found by examining the transition table for the three grid sizes for which data was collected. In particular, the FA models built by the various localization heuristic sizes is of interest. For the 3x3 size grid, with only 2 mines available, the 2x2 localization heuristic had less to learn than on the 4x4 size grid with 3 mines available. When there are only 2 mines in the entire grid, it is impossible for any cell to be adjacent to more than 2 mines. The values of every cell in the grid are limited to being un-clicked, or adjacent to anywhere from 0 to 2 mines. This gives a maximum of 4 different possible states per cell. With 3 mines in the entire grid, the possibilities for each cell increase to include the possibility of a cell being adjacent to 3 mines. Although this includes impossible states, the maximum number of possible states for the 2x2 localization can be no greater than 4^4 with 2 mines and 5^4 with 3. For the 3x3 localization the difference is even worse since the comparison is now the difference between 4^9 and 5^9.

Despite this problem of increasing complexity at the local level, the number of possibilities per cell does not increase indefinitely. Each cell can only be directly adjacent to a maximum of 8 cells. Even with an infinite supply of mines, the number of possible values per cell can be no larger than the possibility of being un-clicked or clicked with a displayed value of anywhere from 0 to 8, giving us a maximum of 10 possibilities per cell. The result is that in a worst case scenario, no matter how large the grid is, the 2x2 localization heuristic would never contain more than 10^4 distinct states since there are 10 possible values per cell and 4 cells in a 2x2 localization. For the 3x3 localization heuristic, the maximum number of states would be 10^9.

Due to the highly localized nature of minesweeper, it is very likely that once a

sufficiently detailed FA model is built for a local area of sufficient size, the overall performance for FABLE will approach the performance level of perfect play. This is due to the fact that in minesweeper, very good local decisions tend to also be very good global decisions. To prove this would require running FABLE on much larger grid sizes with much longer data runs to see if the decrease in the rate of improvement appears to level off after a certain complexity is reached. The performance level that FABLE achieves on these larger grids would then need to be compared to what the performance levels of perfect play for these grids actually are. Since determining what choice constitutes perfect play is NP-Complete [14], proving that FABLE has achieved a level of performance that is near perfect would become intractable as the size of the grid grows. Future research could still be done to at least verify if FABLE always achieves a level that is near perfect for those grid sizes that are tractable though.

4.3 Tic-Tac-Toe

Tic-Tac-Toe is a very old game that is widely known. The game in its standard form has been studied to the point that it is strongly solved, meaning that perfect play is known for both players from every possible position. For more information about tic-tac-toe, as well as many other games, an excellent resource is Victor Allis' Ph.D. thesis, "Searching for Solutions in Games and Artificial Intelligence." [2].

4.3.1 Game Theory

Game Tree Size

Despite the fact that standard tic-tac-toe is a solved problem, the known solution is not valid for the variation of tic-tac-toe that was implemented in the game engine for this thesis. Due to this, I will show how to determine the size of the game tree for the modified version of tic-tac-toe.

Finding the game tree size for the modified tic-tac-toe is straightforward. As with minesweeper, let us define a as the number of squares on the board. On the first turn, player X may choose any of the a possible squares. On the next turn, player O may choose any of the $(a-1)$ squares remaining. With each turn, the number of cells is reduced by one. This means that the game tree size is exactly $a!$.

Probability of Winning

The probability of winning tic-tac-toe by chance is an interesting question for an arbitrarily sized board. Unlike minesweeper, tic-tac-toe is a two-player game. As is most often the case with multi-player games, the final outcome of the game is heavily influenced by the skill of the players. The remainder of this section will be discussing the probabilities of winning under different conditions for the variant of tic-tac-toe used by FABLE. Due to this, the following calculations and arguments are my own.

For the purpose of evaluating the probability of winning from a theoretical sense, we will only consider two skill levels. The first is the skill level of an agent which makes every decision randomly. The second is the skill level of an agent that always makes a perfect choice. Using these two agent types, there are 4 possible types of

matches to consider. They are random X versus random O, perfect X versus random O, random X versus perfect O, and perfect X versus perfect O.

For random X versus random O, we can ignore any need to evaluate what would occur during play. This is because no matter what happens during play, neither agent will use any of the information contained in the current situation to influence its next decision. The result is that all possible final outcomes are equally likely to occur. Finding the probability that X wins, O wins, or that there is a tie is done by counting the number of final board positions that correspond to that outcome and dividing it by the number of possible outcomes. The total number of possible outcomes is:

$$\binom{a}{\lfloor \frac{a}{2} \rfloor} \tag{4.9}$$

Determining if a final board position is a win for X, O, or a tie can easily be done in polynomial time. The problem is that the number of board positions grows factorially as the size of the board increases. This means that determining the expected outcome by explicitly counting the possibilities is intractable. Despite this, a few properties can still be determined for arbitrarily large boards.

The first case to consider is when the size of the board results in a value for a that is even. In this case, for every final board position that results in a win for X, there is a final board position that reverses the position of the X's and O's precisely. Thus, there must be equal number of ways for X to win as for O. When a is even, a random agent playing against another random agent has exactly the same odds of winning as its opponent. The only unknown in this case is how often the game will end in a tie.

The second case to consider is when a is odd. This tilts the balance in favor of

player X. The reason is that with an odd number of squares, player X will get to place one more X on the board than the number of O's that player O will be allowed to place. The presence of an additional X on the board increases the number of ways that player X can complete 3 in a row compared to player O. For two random agents playing the game, this will result in player X having a higher probability of winning than player O.

Now we can consider the case where a perfect agent is playing against a random agent. To calculate the exact probability for each of the 3 possible outcomes of the game would require an algorithm to find the perfect move. The perfect move depends on what the anticipated counter move will be by the other player. Normally, perfect play for a multi-player game is considered to be finding the move that is most likely to lead to a win, or at least avoid defeat, even if the opponent is expected to make a perfect counter move. A straightforward way of determining this is to look at all possible moves and evaluate the perfect counter move for each one. Of course, the way to determine the perfect counter move will require evaluating all of the perfect counter counter moves. The method for determining perfect play is therefore a recursive algorithm that is intractable for all but the smallest of boards.

Since finding the perfect move is intractable, it is also intractable to determine the exact probability of any outcome. This does not mean that certain characteristics of the results of perfect play can not be determined. As a substitute for perfect play, we can consider specific strategies that are tractable and which allow at least some properties of perfect play to be proven. For instance, if a strategy exists that allows player X to force a tie under all circumstances, then perfect play should be able to

at least match that strategy in its effectiveness. Perfect play may be able to actually allow player X to win instead of tie, but perfect play will certainly not allow X to lose when there is a known strategy to prevent a loss.

A strategy that is tractable and allows some conclusions to be drawn for arbitrarily large boards is a mirror move, or symmetry strategy. Mirroring an opponent's moves is a strategy frequently used in two player games. The basic idea is that mirroring your opponent's move will generally give you about the same advantages and disadvantages as your opponent. With the variation of tic-tac-toe that we are considering, this strategy is sufficiently powerful to allow several properties of perfect play to be deduced.

Let us consider the case where a is even. Further suppose that for every move that player X makes, player O chooses the square that is the mirror image of the one X chose. At the end of the game, the arrangement of X's and O's will be perfectly symmetrical. This symmetry ensures that whatever final score X has, O has the same score, resulting in a tie. If at some point during the game, player O decides to make a move that is not a mirror image, then player X can make the mirror image choice of player O. When X mirrors O, the last possible choice for player O before the board is filled will be the choice that restores perfect symmetry. This demonstrates that player X never has to lose when a is even. The only reason for perfect play for player X to deviate from using symmetry is if there is a way to force a win.

Now we can consider the case where a is odd. Any board where a is odd will have a single square in the center. When the center square is clicked on this type of board, the other player can not click on the mirror image since the mirror image of

the center square is the center square itself. If player X chooses to click on the center square first, player O will be forced to pick a square which is not the mirror image of what X chose. Thereafter, player X can always mirror any choice player O makes until the game is over. At this point, the perfect symmetry of the board except for the center square will ensure that for every point O gained during the game, X will have a corresponding point. In addition, player X might have additional points in the event that any 3 X's in a row were completed that utilize the center square. This proves that for odd size boards, perfect play for player X can force a tie at worst. It might be possible for player X to deviate from the mirroring strategy to force a win no matter what player O does, but a proof for how to do so is lacking.

Determining the outcome for perfect play by player O is simple now that perfect play for player X has been established. For boards where a is even, player O can at least force a tie. If player X makes a mistake, then perfect play for player O will result in a win for player O. For boards where a is odd, player O may or may not be able to force a tie. For a 3x3 board, it can be proven that player O can force a tie by exhaustion. It is unclear if that result generalizes to larger boards.

The last case to consider is what happens when both players play perfectly. Where a is even, neither player X or O can force a win. This means that perfect play will always result in a tie. When a is odd the result is unclear. For the 3x3 board it is known that player O can force a tie no matter what strategy player X employs. For larger boards it is unclear whether or not this will remain true. If player X can force a win no matter what player O does, then the result will be that player X will win every time and player O will lose. If player O can force a tie no matter what player

X does, then the result will be that the game will always end in a tie. Since player X can force a tie, player O can not win.

4.3.2 Data Results

The data for tic-tac-toe was generated for 3x3, 4x4, and 5x5 size boards. For each board size, 4 separate data runs were made. The first was a random agent as X versus a random agent as O. The second run was a FABLE agent as X versus a random agent as O. The third run was a random agent as X and a FABLE agent as O. The last run was a FABLE agent as X versus a FABLE agent as O. In each case, 1000 games were played. The data was then grouped into 10 sets of 100 games each.

The expected results for the random X versus random O are that the data should agree with the theory given above. That is, for the 3x3 and 5x5 boards, the data should show at least some degree of bias in favor of X winning. For the 4x4, the expected result is that X and O should both win about the same amount.

To evaluate the expected results for the FABLE agents, we need to consider that at first the agent knows nothing about the game and will pick at random. As the FABLE agent plays, it should learn more about the game and begin to make choices in a less random fashion. At best, the FABLE agent will learn enough to play perfectly. In practice that is unlikely. The expected result is that as the FABLE agent learns, its performance will improve versus making choices at random. The theory regarding perfect agents that was given earlier serves to establish a ceiling on the performance that is possible for FABLE to achieve.

The results for the random versus random on the 3x3, 4x4, and 5x5 boards are

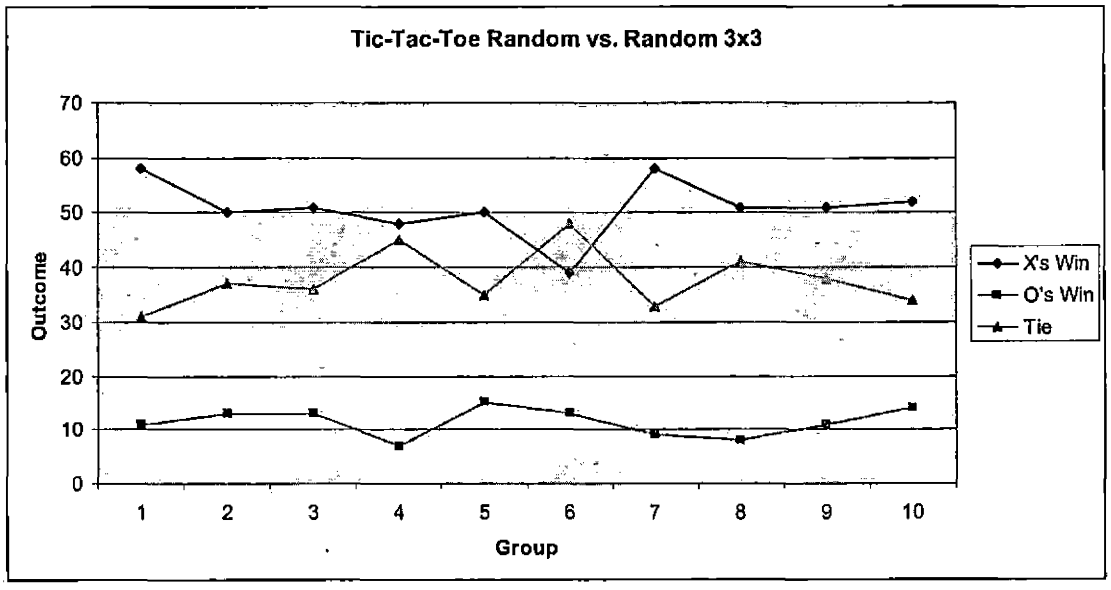

Fig. 4.7: Results of Random versus Random on Tic-Tac-Toe 3x3.

found in table 4.3. Charts with the data points plotted have also been prepared and are found in figures 4.7, 4.8, and 4.9.

The results for the random data follow the expectations that were established during the theoretical analysis of the game. This can be seen by looking at each board size, determining the theoretical predictions for that board, and then comparing the actual result to those predictions. The 3x3 and 5x5 boards are cases where a is odd. Theory predicts that these boards should favor of X over O. The results show that for the 3x3 board, X wins 50.8% of the time while O wins 37.8%. For the 5x5 board it is 62.4% and 26.9% respectively. Both results support the theory that random play favors X in these cases. For the 4x4 board, the value of a is even. In this case theory predicts that X and O should have the same probability of winning. The results show that X wins with an average of 39.6% and O wins 38.2%. The standard deviations for both X and O put the difference between the averages to be much less than one

Tab. 4.3: Results of Random versus Random on Tic-Tac-Toe.

Results of Random Versus Random on Tic-Tac-Toe									
Group	3 x 3			4 x 4			5 x 5		
	X	O	Tie	X	O	Tie	X	O	Tie
1	58	11	31	39	39	22	62	28	10
2	50	13	37	38	43	19	59	31	10
3	51	13	36	37	43	20	67	23	10
4	48	7	45	43	34	23	62	31	7
5	50	15	35	34	45	21	62	30	8
6	39	13	48	49	33	18	70	20	10
7	58	9	33	38	34	28	62	27	11
8	51	8	41	38	36	26	55	30	15
9	51	11	38	39	40	21	66	23	11
10	52	14	34	41	35	24	59	26	15
Average	50.8	11.4	37.8	39.6	38.2	22.2	62.4	26.9	10.7
Min	39	7	31	34	33	18	55	20	7
Max	58	15	48	49	45	28	70	31	15
σ	5.31	2.67	5.39	4.06	4.39	3.12	4.35	3.84	2.58

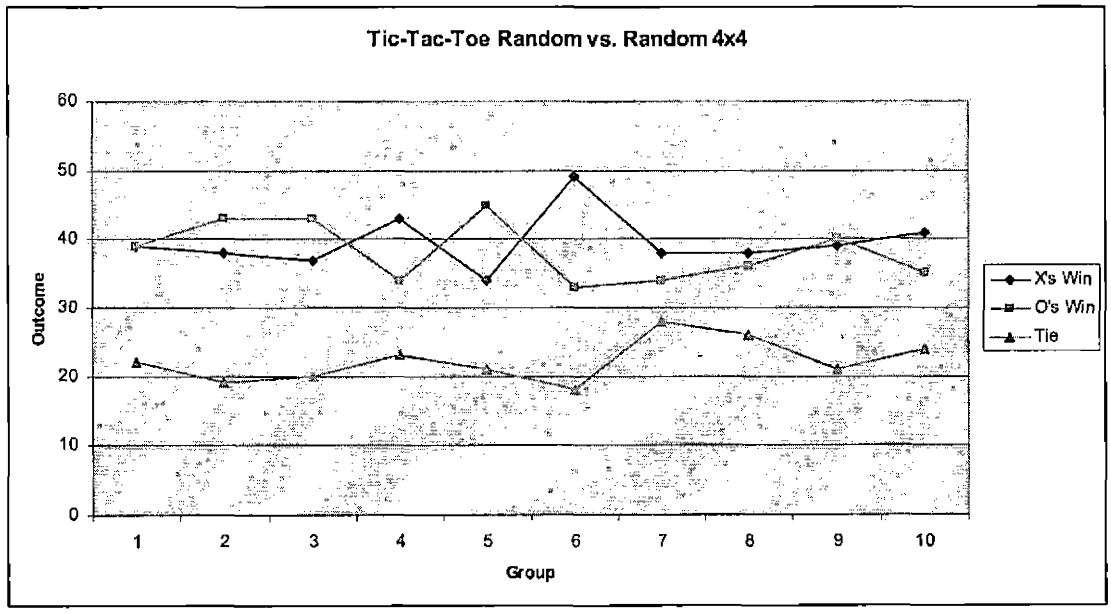

Fig. 4.8: Results of Random versus Random on Tic-Tac-Toe 4x4.

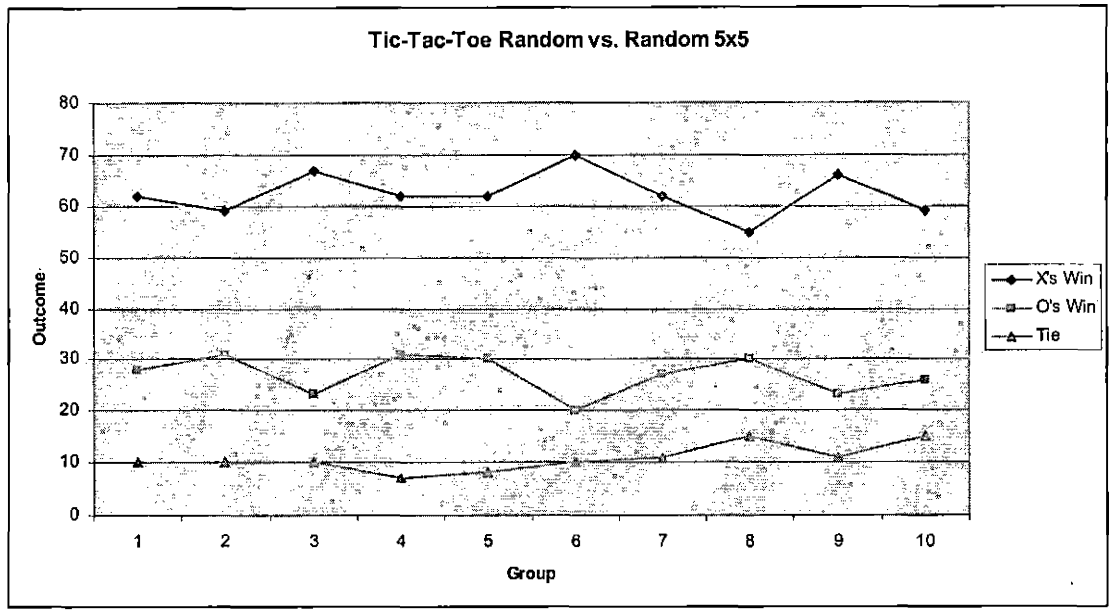

Fig. 4.9: Results of Random versus Random on Tic-Tac-Toe 5x5.

Tab. 4.4: Results of FABLE versus Random on Tic-Tac-Toe.

Results of FABLE Versus Random on Tic-Tac-Toe									
Group	3 x 3			4 x 4			5 x 5		
	X	O	Tie	X	O	Tie	X	O	Tie
1	59	8	33	61	21	18	80	16	4
2	61	1	38	68	12	20	82	8	10
3	57	1	42	70	14	16	92	5	3
4	54	1	45	76	13	11	93	3	4
5	58	1	41	79	6	15	93	3	4
6	54	0	46	76	12	12	95	3	2
7	54	0	46	79	3	18	93	3	4
8	49	0	51	73	8	19	91	3	6
9	56	1	43	67	11	22	89	3	8
10	54	0	46	74	11	15	90	5	5

standard deviation, which supports the proposition that the odds of either winning are the same.

With the baseline performance for random agents established, we can now look at the data for when a FABLE agent plays as X against a random agent for O. The results can be found in table 4.4. The charts are also available in figures 4.10, 4.11, and 4.12.

The results for FABLE as player X versus a random agent for player O are very impressive on the 3x3 board. The data show that for this case, by the second set

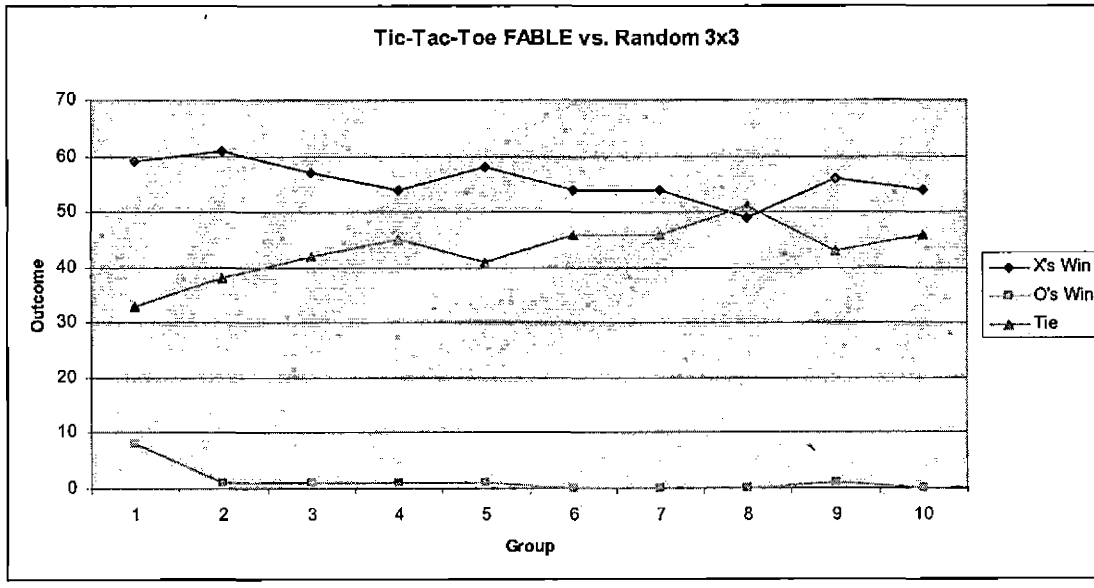

Fig. 4.10: Results of FABLE versus Random on Tic-Tac-Toe 3x3.

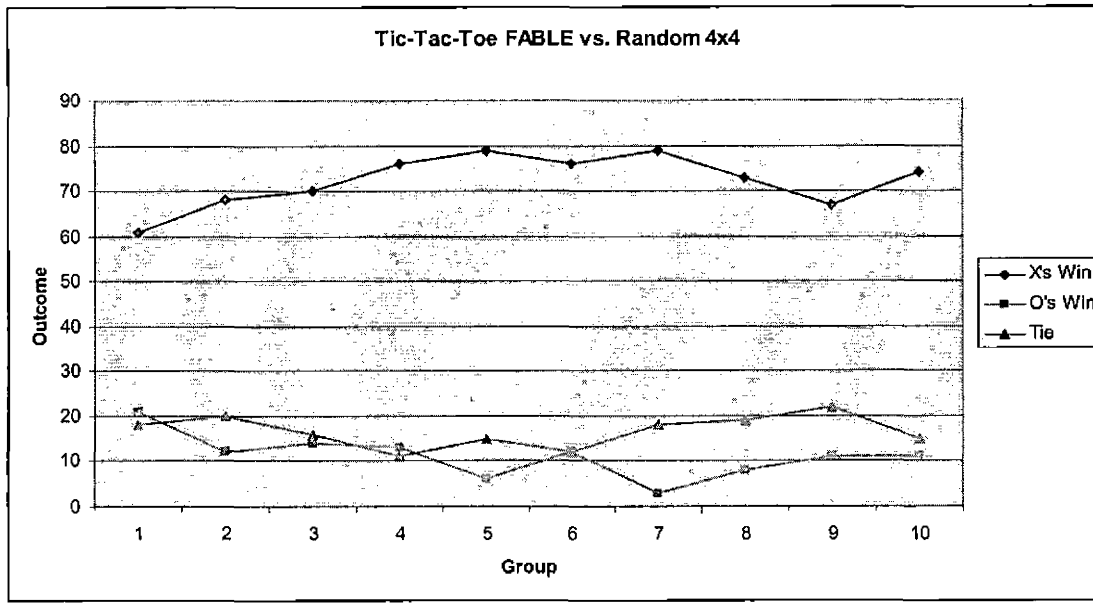

Fig. 4.11: Results of FABLE versus Random on Tic-Tac-Toe 4x4.

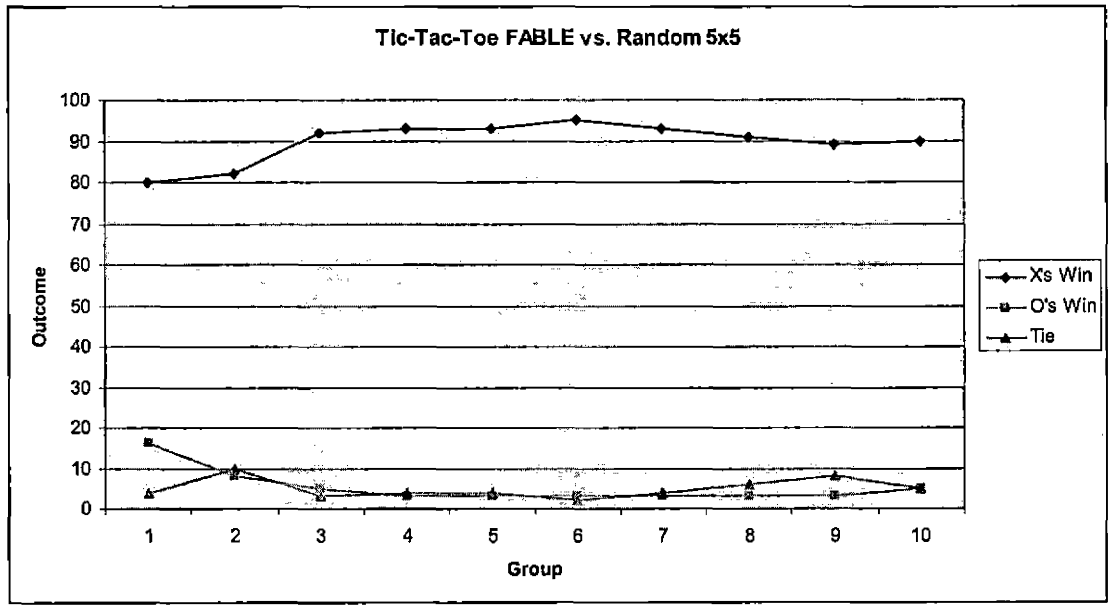

Fig. 4.12: Results of FABLE versus Random on Tic-Tac-Toe 5x5.

of 100, player O never wins more than 1 game per 100 ever again. The drop in the average of player O winning is clearly the result of the FABLE agent learning how to block player O from winning. In fact, FABLE learned to play so well that player O only won 1 of the last 500 games. As for the probability of player X winning, the data do not show any clear improvement. The predicted result for perfect play for X is to win only if player O makes a mistake. Perfect play will only result in a tie otherwise. The results for the last 500 games look very similar to this prediction, although not quite perfect. Due to this, it would appear that on the 3x3 board, FABLE learned to play at a level that is very close to perfect.

For the 4x4 board a significant shift in the distribution of game outcomes is apparent within the first 100 games. The rate at which player X won during the first 100 games was actually 5 standard deviations greater than the expected outcome for a random agent. Throughout the rest of the series, the rate at which X won continued

to show a slight upward trend. In addition, the ratio of wins for O to the number of ties showed a clear downward trend. This means that on the 4x4 board, the FABLE agent learned how to increase the odds of X winning while also learning how to decrease the odds of O winning when X does not win.

For the 5x5 board the FABLE agent was able to improve the number of wins for X in the first set of 100 games to a level that is 4 standard deviations above the expected results for two random agents playing against each other. In addition to the large improvement over random play during the first 100 games, there is another surge in performance during the third set of 100 games. During the third set, the number of wins for X jumps to being greater than 6 standard deviations above the expected results for random play and then sustains that level of performance for the remainder of the data run.

Once the results for FABLE as player X against a random agent as player O were completed. Data was generated for the case where player X was random and player O was a FABLE agent. The results of this data run are found in table 4.5. As done in the previous cases, charts of the result for each of the board sizes have also been prepared and can be found in figures 4.13, 4.14, and 4.15.

The data for the 3x3 board of Random X versus FABLE O show the most continuous and consistent trends of any of the data runs for tic-tac-toe. In this case, an extremely clear upward trend for the rate at which O wins is seen for the entire data set with the exception of the last data point. The rate at which X wins shows a very clear downward trend that is mostly smooth. The rate at which a tie resulted showed no clear trend. Perfect play for O would result in every game either being a win for

Tab. 4.5: Results of Random versus FABLE on Tic-Tac-Toe.

Results of Random Versus FABLE on Tic-Tac-Toe									
Group	3 x 3			4 x 4			5 x 5		
	X	O	Tie	X	O	Tie	X	O	Tie
1	52	16	32	32	43	25	57	34	9
2	47	19	34	27	55	18	41	57	2
3	49	23	28	14	62	24	35	56	9
4	42	27	31	14	68	18	36	54	10
5	35	33	32	14	58	28	28	58	14
6	40	39	21	15	67	18	35	56	9
7	31	40	29	14	66	20	30	59	11
8	37	44	19	14	66	20	29	59	12
9	34	52	14	12	65	23	30	55	15
10	32	43	25	16	69	15	26	60	14

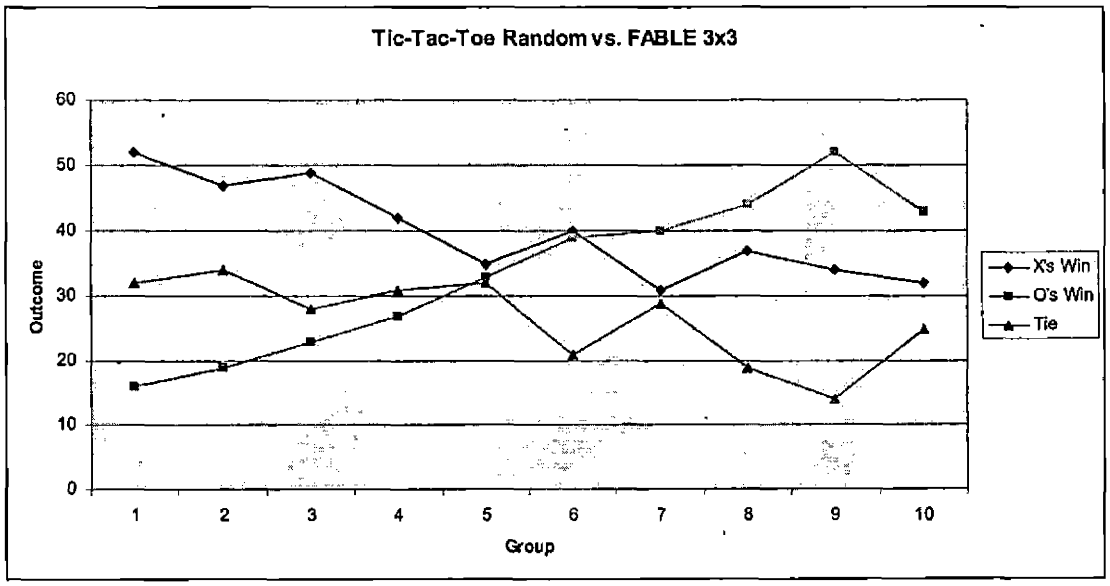

Fig. 4.13: Results of Random versus FABLE on Tic-Tac-Toe 3x3.

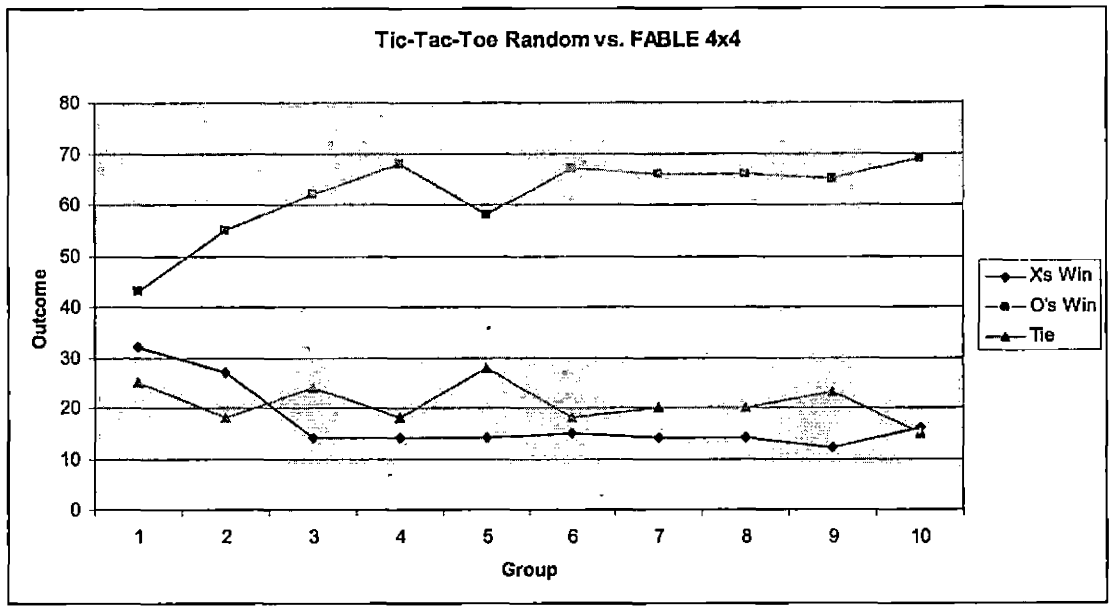

Fig. 4.14: Results of Random versus FABLE on Tic-Tac-Toe 4x4.

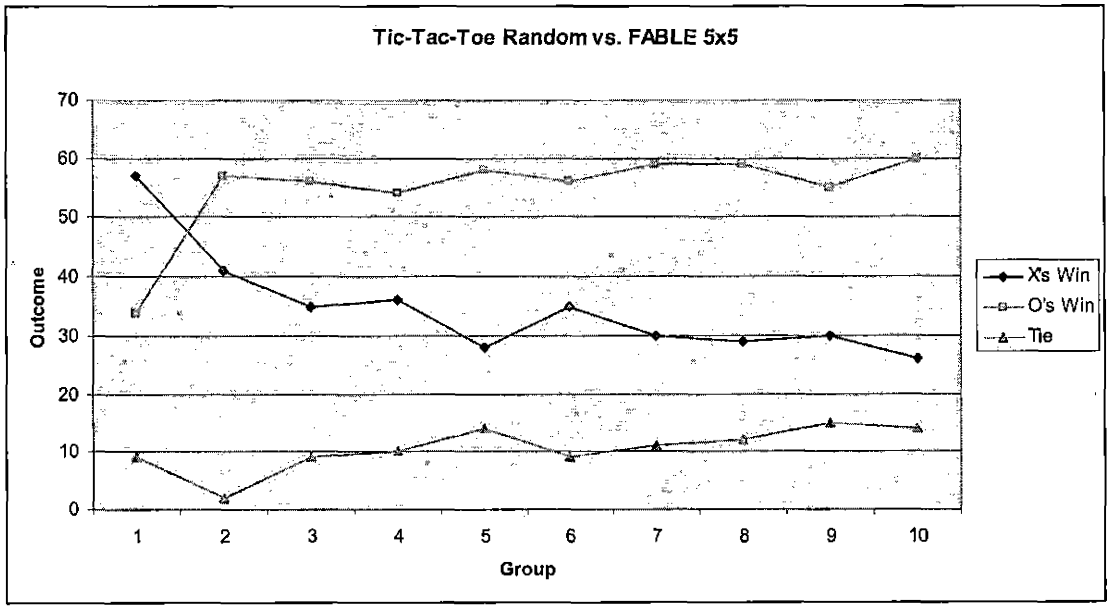

Fig. 4.15: Results of Random versus FABLE on Tic-Tac-Toe 5x5.

O or a tie. Since the rate at which X won showed a clear downward trend, it is safe to say that FABLE showed an increasing skill level as it played.

The 4x4 board for random X versus FABLE O shows an improving rate of O winning for the first 400 games. After that, there is no clear upward or downward trend for the rate at which O wins. There is a similar downward trend for the rate at which X wins which also appears to level off after the first 400 games. The win rates for X and O are very stable until the last set of 100 games. For the last data point, O wins the highest number of games seen during the entire data run. It is possible that this data point represents the beginning of another shift in the probabilities of the outcomes, but without more data it is impossible to draw any definitive conclusions.

For the 5x5 board with random X versus FABLE O, a pattern that is somewhat similar to the 4x4 board is seen. In this case, the performance gain at the beginning happens faster, but it also levels off sooner. Once again, the data show a leveling

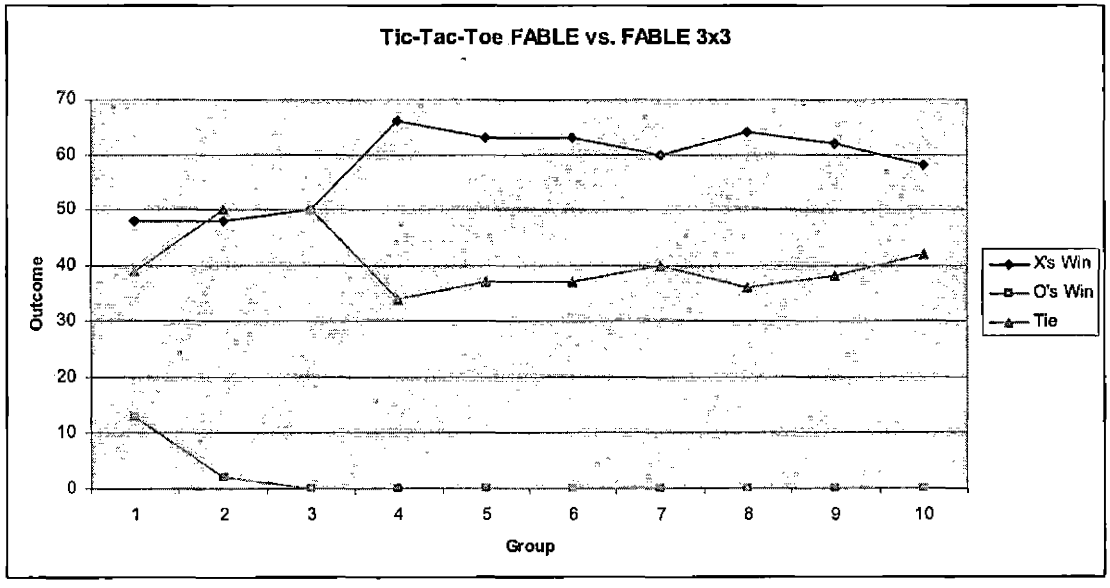

Fig. 4.16: Results of FABLE versus FABLE on Tic-Tac-Toe 3x3.

off for the win rates of X and O that are better than random but less than perfect. Interestingly, the last data point for the 5x5 is also one which shows a variation from the average values that might be indicative of FABLE having learned enough to cause another shift in the probabilities of the outcomes similar to the gain seen at the beginning. Despite that, the data point is not so far away from the established performance levels as to make it clear whether the increase is due to increased skill or luck.

With the data where a random agent plays as X against a FABLE agent as O complete, we can now turn our attention to the last combination of agent types to be tested. For FABLE X versus FABLE O, the results for all 3 board sizes are found in table 4.6. The charts are found in figures 4.16, 4.17, and 4.18.

For the scenario where FABLE plays against FABLE it is hard to say what the expected results are. In this case, both FABLE agents will have access to the same

Tab. 4.6: Results of FABLE versus FABLE on Tic-Tac-Toe.

Results of FABLE Versus FABLE on Tic-Tac-Toe									
Group	3 x 3			4 x 4			5 x 5		
	X	O	Tie	X	O	Tie	X	O	Tie
1	48	13	39	62	14	24	87	9	4
2	48	2	50	70	11	19	91	7	2
3	50	0	50	74	12	14	93	4	3
4	66	0	34	68	14	18	86	8	6
5	63	0	37	63	22	15	85	8	7
6	63	0	37	73	13	14	87	3	10
7	60	0	40	67	19	14	81	14	5
8	64	0	36	61	30	9	90	5	5
9	62	0	38	68	20	12	87	6	7
10	58	0	42	62	16	22	92	6	2

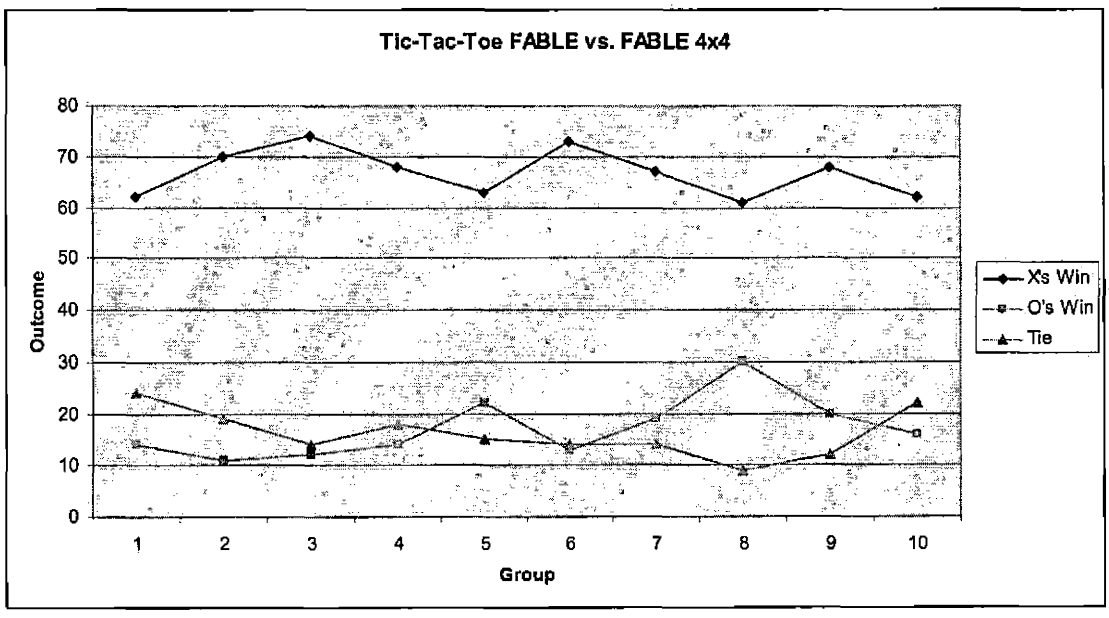

Fig. 4.17: Results of FABLE versus FABLE on Tic-Tac-Toe 4x4.

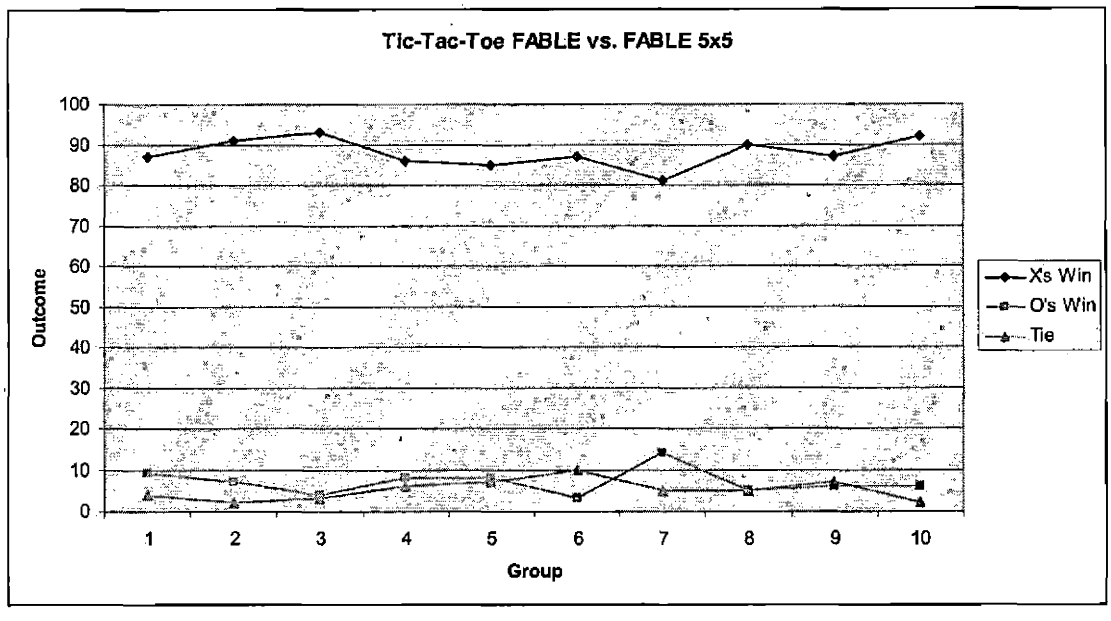

Fig. 4.18: Results of FABLE versus FABLE on Tic-Tac-Toe 5x5.

data and will both use the exact same algorithm for deciding what to do. This does not imply that both agents will have the same skill level, but they should both have at least the same capacity to increase in skill level.

The data for the 3x3 with FABLE versus FABLE show that the results for the first set of 100 are indistinguishable from a data point in the random versus random data. Following that first data point, the rate at which O wins quickly drops to 0. In fact, O wins none of the last 800 games. For the remainder of the test run, the rate at which X wins takes a jump initially, but appears to slowly wane thereafter. The trend at the end of the data run is that O never wins, while X wins at a decreasing rate. This corresponds to an increasing rate for the game resulting in a tie. By the end of the data run, the results look much more like the predictions of perfect play versus perfect play than the predictions of random versus random play.

The 4x4 board is very interesting. For this size board, the theory is that X and O should win with the same frequency when played randomly. For perfect agents, the expected result is for every game to end with a tie. In both extremes of skill level, the expected result is that X and O are evenly matched. That is why it is interesting that the results show that X wins about 70% of the time for the entire data run. Granted, there appears to be a slight downward trend in the rate at which X wins. If that downward trend were to continue long enough, the results would eventually resemble perfect versus perfect play, but it is hard to say if that would indeed occur. It is also unclear why X appears to gain such a strong advantage in the first place.

For the 5x5 board, the results show a very clear bias in favor of X. The only way to see that both agents are learning in this case is to compare the results to those of

FABLE X versus random O. For FABLE X versus random O, O won about 3% of the time. For FABLE X versus FABLE O, O maintains a win rate that is closer to 7%. This is indicative that FABLE O is doing better than random O. From theory, whether or not perfect play for X can force a win for X is still an open question. Since the result of perfect play is unknown, it is difficult to say if the final results of FABLE X versus FABLE O are showing any tendency towards perfection, but either way, both agents do show the ability to perform better than random.

In summary, the data for the various combinations of random and FABLE agents on tic-tac-toe support the proposition that FABLE does learn over time. Clear improvements in capability are seen within the first 100 games in all cases except one. Even for this exception, it is still likely that improvements in skill were occurring since it was a FABLE X versus FABLE O game. It is highly likely that the increase in skill for FABLE X was simply getting offset by the increase in skill for FABLE O, and the absolute skill levels were still low enough that the results still appeared similar to random.

As far as the pattern of learning is concerned, some of the data runs showed improvements in performance that were more or less continuous. Other data runs had a tendency to show occasional spurts in performance that are indicative of step-wise learning. The occasionally non-continuous changes in performance are not inconsistent with the way FABLE learns. It is possible that the spurts occur after a game wherein a critical transition is learned enables a permanent improvement in skill thereafter. The inability to determine any general equation form that the performance should follow made it difficult to apply regression techniques to more rigorously an-

alyze the results. Despite this lack, the change in performance when a FABLE agent was used compared with the result for a random agent still makes it very clear that FABLE does learn, and that it can do so within a tractable amount of time for the game of tic-tac-toe.

4.4 Checkers

4.4.1 Game Theory

While the measure of complexity used for minesweeper and tic-tac-toe was to use the size of the game tree, this measure is not useful in the case of checkers. The reason is that checkers is a game in which a potentially limitless number of moves may occur due to the possibility of board positions repeating during the same game. The result is to make the size of the game tree infinite. This issue can technically be mitigated by the introduction of new rules that force the game to end, but such rules are not official rules. The official rules do have some methods for forcing a draw to be declared so that the game ends, but those rules involve the judgement of a human referee. The end result is that checkers, at least when played by official rules, has an game tree size that is infinite. As a substitute, the space complexity of checkers is analyzed instead.

Space Complexity

Calculating the exact space complexity of checkers is a difficult problem to solve, but there are methods to estimate the complexity. In "Solving the Game of Checkers" Jonathan Schaeffer and Robert Lake derive a formula to estimate the space complexity

of an 8x8 sized board [19]. They note that although the formula does not give an answer that is perfectly accurate, it is a good approximation.

Since FABLE will be tested on boards of various sizes, I found it necessary to take their formula and generalize it. Their formula contains 6 variables, 5 of which are described as follows: "Let b be the number of black checkers, w the number of white checkers, B the number of black kings, W the number of white kings and f the number of black checkers on the first rank." [19] They describe the formula as being the way to find "[t]he number of positions having n or fewer pieces on the board." So the purpose of n is actually contained in the description of what the formula does. Their formula is found in equation (4.10).

$$\sum_{b=0}^{\min(n,12)} \sum_{B=0}^{\min(n,12)-b} \sum_{w=0}^{\min(n-b-B,12)} \sum_{W=0}^{\min(n-b-B,12)} \sum_{f=0}^{\min(b,4)} \mathrm{Num}(b,B,w,W,f) - 1,$$

where

$$\mathrm{Num}(b,B,w,W,f) = \binom{4}{f}\binom{24}{b-f}\binom{28-(b-f)}{w}\binom{32-b-w}{B}\binom{32-b-w-B}{W} \tag{4.10}$$

As we can see, their formula has several constants in it. Their explanation for these constants is that "12 is the maximum number of pieces per side, 32 is the number of squares on the board that can be occupied, 28 is the number of squares that it is legal to place a checker on (checkers become kings on the last rank) and 4 is the number of squares on the black's first rank (these squares cannot hold any white checkers)." [19]

Generalization of this formula is done by taking the constants which were calcu-

lated based on an 8x8 board, and re-writing them as variables based on the dimensions of the board. For a NxM board, let N be the number of rows (or ranks) and M be the number of columns (or files). The first constant to convert is 12, "the maximum number of pieces per side."[19] The maximum number of pieces per side is the same as the number of pieces per side at the start of the game. Due to how the FABLE game engine starts the game, this is the same as the number of squares that can be occupied per row, times the number of rows starting from the bottom, until the row preceding the middle row is reached. Let us call this p. The next constant to consider is 32. It is "the number of squares on the board that can be occupied."[19] Let us call this q. Next we have 28, which "is the number of squares that it is legal to place a checker on."[19] This is the same as the squares that can be occupied for the entire board minus the squares that can be occupied for a single row. Let us call this r. Last we have the constant 4, which "is the number of squares on the black's first rank."[19] Let us call this s. I have given equations that express each of these new variables in terms of N and M in equations (4.11), (4.12), (4.13), and (4.14).

In addition to the constants that were explained, there is one additional constant in the equation which needs to be replaced in terms of N and M. This is 24. No direct explanation for the origin of this constant was offered by Schaeffer and Lake. However, looking at the derivation of the equation, it can be determined what the 24 represents. It is the number of squares that the black checkers which are not on the first rank can occupy. For the 8x8 board this is 24. To see this, consider that there are 32 squares available in total. Of those, the 4 squares on the first rank are excluded by definition, and the 4 squares on the last rank are excluded. The reason

for excluding the latter is because once a regular checker reaches the last rank, it becomes a king. Let us call this t, the equation for which can be found in equation (4.15).

Substituting these values into equation (4.10), I define a formula that finds the space complexity checkers on a NxM sized board. This equation is found in equation (4.16). I have used this formula to find the approximate space complexities for various board sizes. These values can be found in table 4.7. It is important to note that the table only includes even values for N and M. This is because the generalized equation is derived based on the way the FABLE game engine starts a game. Since the game engine only accepts even numbers, the equation is only valid for even numbers.

$$p = \frac{M}{2} * (\frac{N}{2} - 1) \tag{4.11}$$

$$q = \frac{N * M}{2} \tag{4.12}$$

$$r = \frac{N * M}{2} - \frac{M}{2} \tag{4.13}$$

$$s = \frac{M}{2} \tag{4.14}$$

$$t = \frac{N * M}{2} - M \tag{4.15}$$

$$\sum_{b=0}^{\min(n,p)} \sum_{B=0}^{\min(n,p)-b} \sum_{w=0}^{\min(n-b-B,p)} \sum_{W=0}^{\min(n-b-B,p)} \sum_{f=0}^{\min(b,s)} \mathrm{Num}(b, B, w, W, f) - 1,$$

where

$$\mathrm{Num}(b, B, w, W, f) = \binom{s}{f}\binom{t}{b-f}\binom{r-(b-f)}{w}\binom{q-b-w}{B}\binom{q-b-w-B}{W} \tag{4.16}$$

Tab. 4.7: Approximate Space Complexity for Various Sizes of Checkers.

Approximate Space Complexity for Checkers on a NxM Board			
	4	6	8
4	6087	854727	130449919
6	10354431	63838220543	421609358479359
8	11616684159	2353833522293759	500995484682338672639

Probability of Winning

The game of checkers is so complex that I have been unable to find any tractable method of calculating the expected probabilities of outcomes for random agents from an ab initio perspective. I am also unaware of any literature dealing with determining these values in a theoretical sense. For this reason, the establishing of what the performance of two random agents playing against other looks like will be determined purely empirically in the data results section.

The problem of determining perfect play for checkers was an open problem for a very long time. In "The Complexity of Checkers on an N*N Board", Fraenkel et al. establish that the problem of determining perfect play is PSPACE-complete and EXPTIME-complete. Despite these results, in 2007 Schaeffer et al weakly solved the problem of checkers on an 8x8 board with the publication of "Checkers is Solved" [18]. To solve it, they wrote a program called Chinook that ran for 18 years on an average of 50 desktop computers at any time. The result found that perfect play on both sides results in a tie. It is important to note that checkers was only weakly solved. Which means that perfect play is only known for the scenario where it is followed

from the very beginning of the game. Perfect play from an arbitrary position is still an open question for an 8x8 board.

A search for scientific work showing the results of perfect play for smaller checker board sizes has found nothing. Since the 8x8 board has been solved, the smaller boards should technically be tractable, but still required too much computing power to be tractable for the resources available to me. This is especially the case since most computing resources available to me during the writing of this thesis were dedicated to running FABLE, not to proving properties of game theory.

4.4.2 Data Results

Data for checkers was generated using FABLE to create 4 separate data runs of random red versus random white, FABLE red versus random white, random red versus FABLE white, and FABLE red versus FABLE white. In each case 1000 games were played, and the results were grouped into sets of 100 games each according to the order in which they were played. The data was generated for only two board sizes, the 4x4 and 6x6.

The inherently open-ended nature of checkers resulted in games that lasted for a very large number of turns. This was especially the case for the 6x6. Even though the amount of time spent on each individual turn was certainly a reasonable amount of time. In fact, there is a parameter within FABLE that determines how much time the top level agent gives the heuristic agent processes to respond. This parameter was set to 1 minute, which is the same time constraint that is given to human players in official tournament play [21]. The responses always came back within less than the

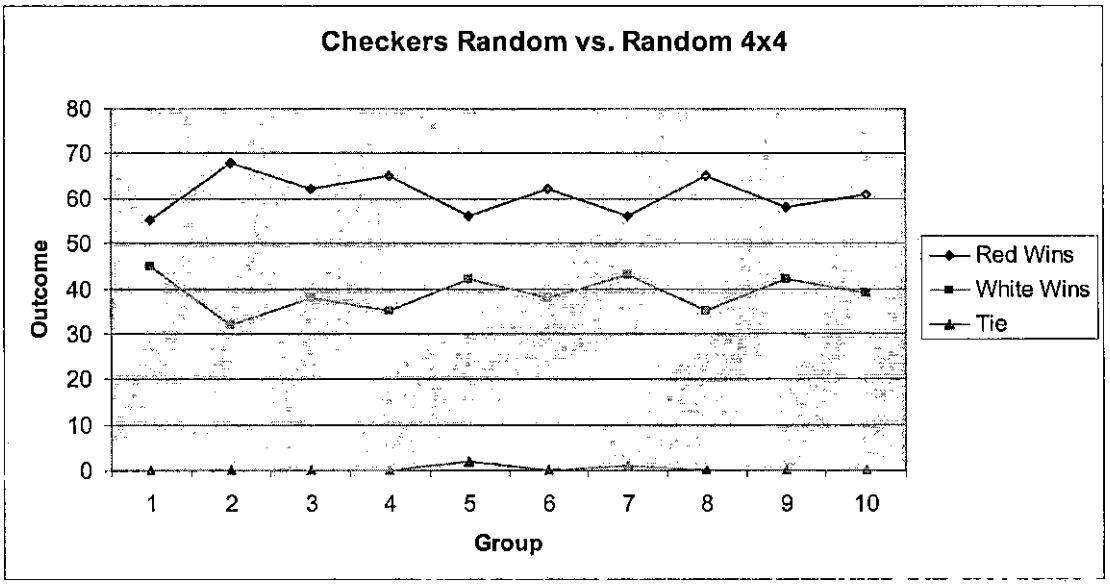

Fig. 4.19: Results of Random versus Random on Checkers 4x4.

allotted time. Despite the relatively good performance for individual turns, the sheer length of each game made generating data for an 8x8 board a task which will be left for future research.

Since there are no theoretical predictions for what the results of random play should look like, the only measure available for random play is the empirical data that was generated during the data runs. This data will be the only benchmark available to judge what improvement FABLE was able to make over random play. These results are found in table 4.8. The charts for the individual board sizes are found in figures 4.19 and 4.20.

The random data suggest that on a 4x4 board, the red player has an advantage. Both the rate at which red wins and the rate at which white wins show a standard deviation of about 4%. With respective win rates of 60.8% to 38.9%, the distance between the averages is clearly about 5 standard deviation. A difference this large

Tab. 4.8: Results of Random versus Random on Checkers.

Results of Random Versus Random on Checkers						
Group	4 x 4			6 x 6		
	Red Wins	White Wins	Tie	Red Wins	White Wins	Tie
1	55	45	0	50	40	10
2	68	32	0	43	49	8
3	62	38	0	43	51	6
4	65	35	0	56	36	8
5	56	42	2	51	42	7
6	62	38	0	46	47	7
7	56	43	1	37	59	4
8	65	35	0	48	44	8
9	58	42	0	37	59	4
10	61	39	0	54	42	4
Average	60.8	38.9	0.3	46.5	46.9	6.6
Min	55	32	0	37	36	4
Max	68	45	2	56	59	10
σ	4.44	4.12	0.67	6.55	7.72	2.07

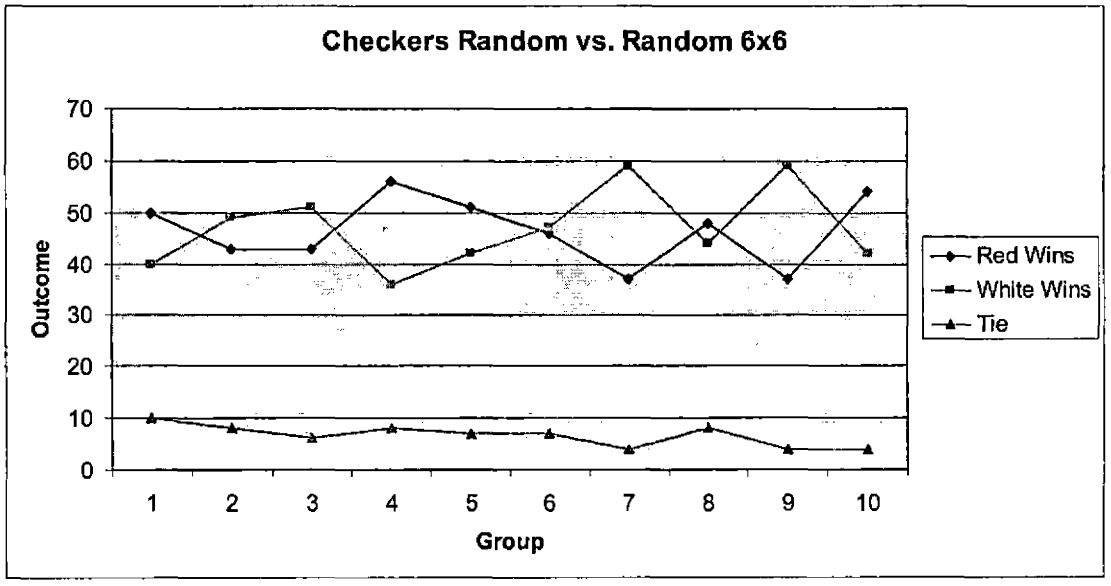

Fig. 4.20: Results of Random versus Random on Checkers 6x6.

makes it unlikely that the apparent advantage to red was merely due to luck.

For the 6x6 board the win rates for red and white are 46.5% and 46.9% with a standard deviation for both that is greater than 6%. This strongly suggests that on a 6x6 board, neither player has an advantage over the other, or that if there is an advantage, it is not as strong as was the case on 4x4.

Next we will examine the data for FABLE red versus random white. These results are found in table 4.9. The charts for the individual board sizes are found in figures 4.21 and 4.22.

For FABLE red versus random white, the 4x4 board data clearly establish a performance level that is significantly better than random versus random. By the first data point, which corresponds to the first 100 games, the FABLE agent has already increased the win rate to 90%. No sustained improvement or reduction in capability is observed throughout the remainder of the data although the degree of variation

Tab. 4.9: Results of FABLE versus Random on Checkers.

Results of FABLE Versus Random on Checkers						
Group	4 x 4			6 x 6		
	Red Wins	White Wins	Tie	Red Wins	White Wins	Tie
1	90	8	2	53	39	8
2	90	10	0	56	34	10
3	92	8	0	61	32	7
4	90	9	1	65	30	5
5	94	6	0	58	36	6
6	90	9	1	68	28	4
7	87	13	0	55	33	12
8	94	5	1	60	36	4
9	88	11	1	53	40	7
10	94	5	1	53	40	7

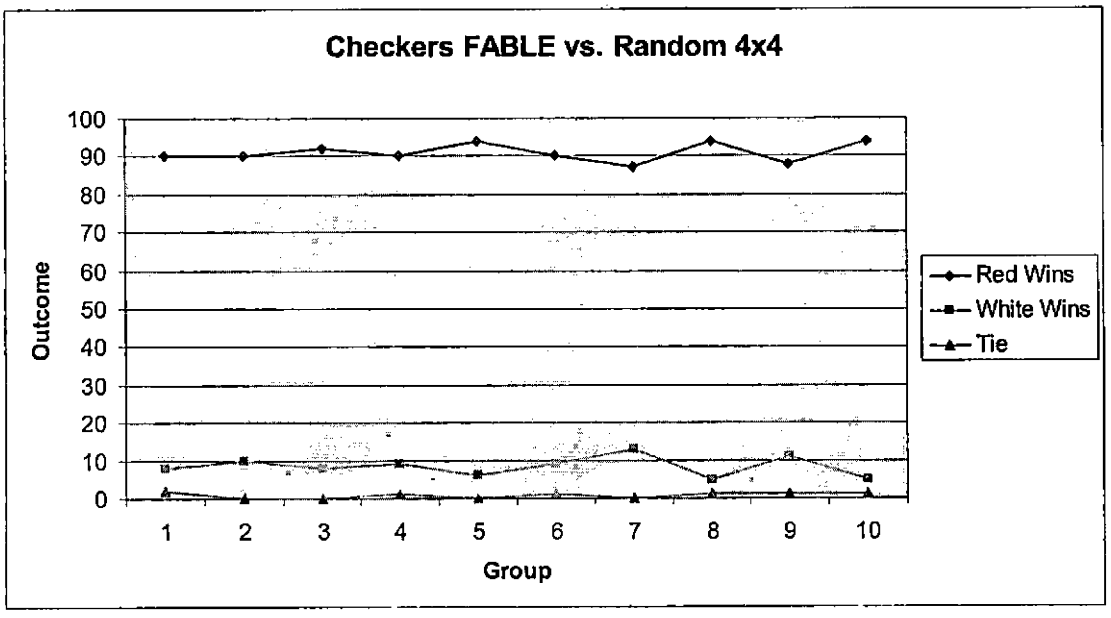

Fig. 4.21: Results of FABLE versus Random on Checkers 4x4.

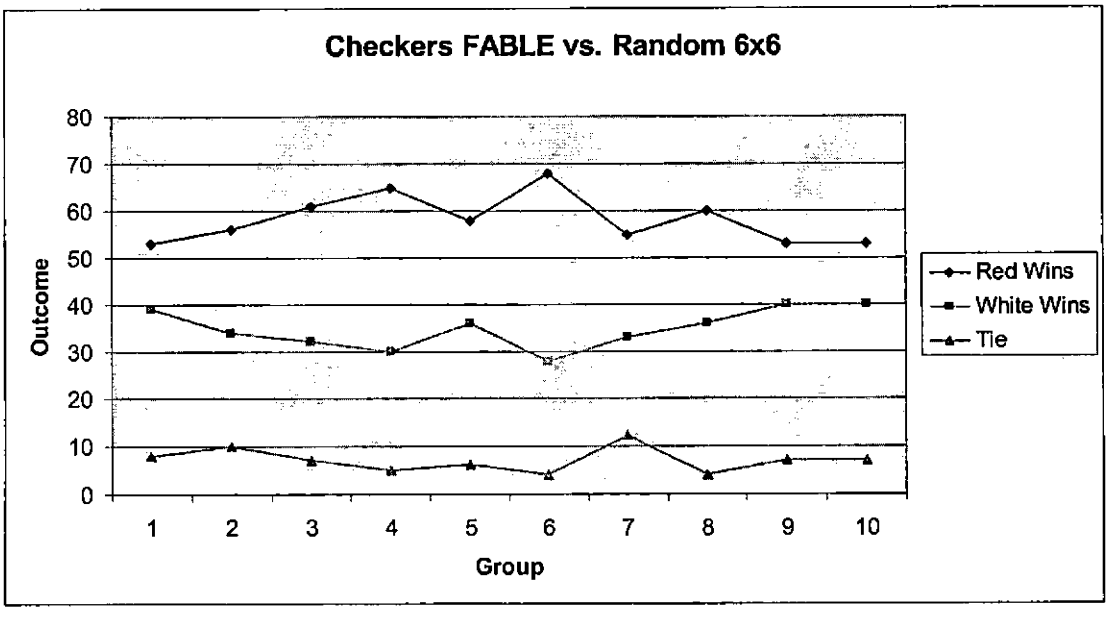

Fig. 4.22: Results of FABLE versus Random on Checkers 6x6.

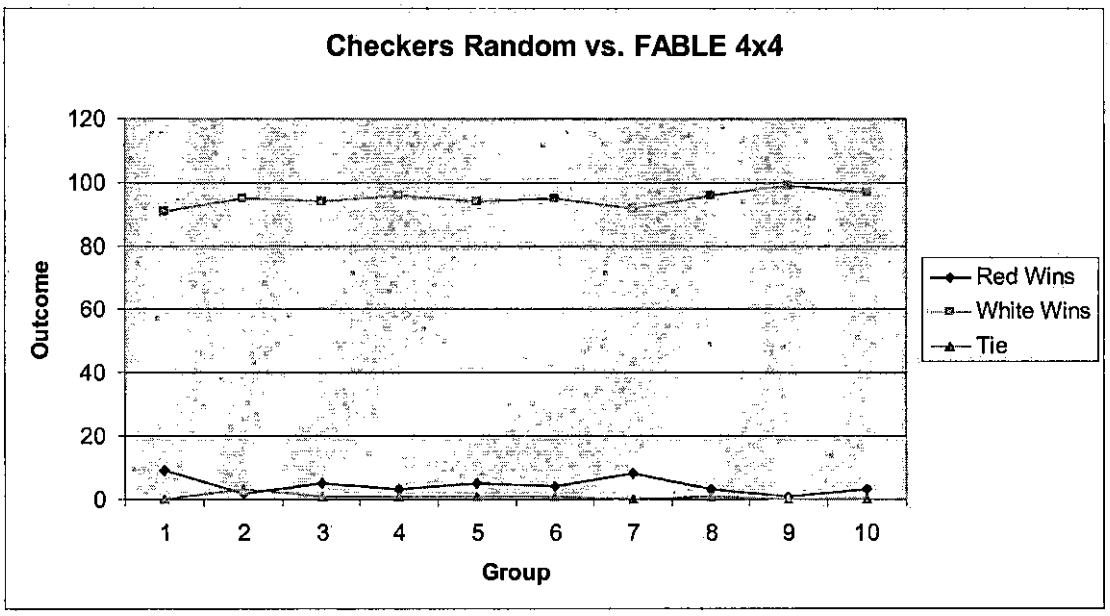

Fig. 4.23: Results of Random versus FABLE on Checkers 4x4.

appears to be increasing towards the end.

On the 6x6 board, FABLE red versus random white appears to follow a sort of bell curve. Throughout the entire data run the FABLE agent maintains a win rate that is higher than the random data, which indicates some learning occurred. What is hard to understand is why performance climbs and then drops. It would be interesting to see what happens if a longer data run is permitted.

Now we will examine the data for random red versus FABLE white. These results are found in table 4.10. The charts for the individual board sizes are found in figures 4.23 and 4.24.

Starting with the 4x4 board, the results of random red versus FABLE white look very much like the inverse of the results for FABLE red versus random white. The primary difference is that with FABLE as white, the win rate hovers around 95% compare to a little over 90% with FABLE as red. The more interesting observation

Tab. 4.10: Results of Random versus FABLE on Checkers.

Results of Random Versus FABLE on Checkers						
Group	4 x 4			6 x 6		
	Red Wins	White Wins	Tie	Red Wins	White Wins	Tie
1	9	91	0	27	69	4
2	2	95	3	33	59	8
3	5	94	1	30	63	7
4	3	96	1	41	55	4
5	5	94	1	36	63	1
6	4	95	1	35	63	2
7	8	92	0	32	59	9
8	3	96	1	39	55	6
9	1	99	0	27	67	6
10	3	97	0	32	65	3

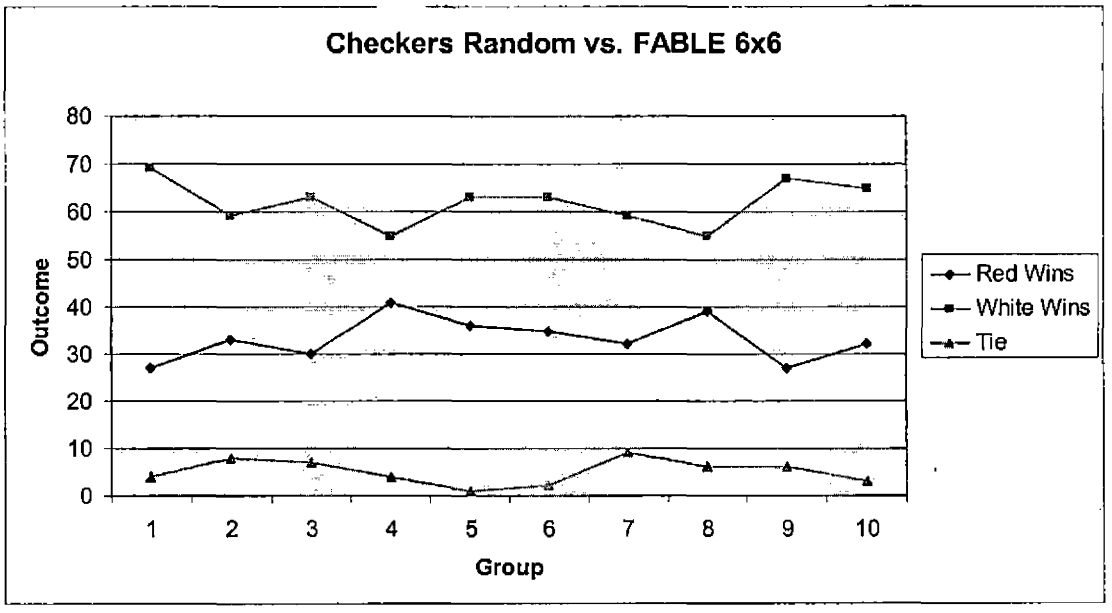

Fig. 4.24: Results of Random versus FABLE on Checkers 6x6.

is that during the random testing, it was seen that the 4x4 appears to strongly favor red. Despite that finding, FABLE white was able to achieve a performance that was even better than FABLE red. This suggests while a 4x4 board may favor red when played randomly, when played intelligently it may be that it does not.

The 6x6 board for random red versus FABLE white shows a clear increase in the win rate for white over purely random play which does establish that some learning did occur. The data here though are just as hard to understand as the data for the case where FABLE was red and random was white. In one, the performance climbs and then drops, in the other it drops and then climbs back up. In both cases the win rate for FABLE remains clearly higher than purely random play, but I have no explanation for why the performance changes direction during the middle of the data run. Even stranger is that the peak and trough both occur during the fifth and sixth data points.

Tab. 4.11: Results of FABLE versus FABLE on Checkers.

Results of FABLE Versus FABLE on Checkers						
Group	4 x 4			6 x 6		
	Red Wins	White Wins	Tie	Red Wins	White Wins	Tie
1	5	95	0	43	44	13
2	8	92	0	48	44	8
3	4	96	0	39	51	10
4	4	95	1	42	41	17
5	6	94	0	45	43	12
6	5	95	0	38	53	9
7	5	95	0	44	43	13
8	7	93	0	43	49	8
9	5	95	0	39	50	11
10	10	90	0	47	37	16

Moving on to the results for FABLE red versus FABLE white, we can find the table in 4.11. The charts are in figure 4.25 and 4.26.

Looking at the results of the data for the 4x4, we can see that FABLE red versus FABLE white is almost impossible to distinguish from the results of random red versus FABLE white. This makes it appear that on a 4x4 board random play favors red, but when played more intelligently it favors white.

The results for FABLE red versus FABLE white for the 6x6 are similar to the results for random versus random. In both cases the data points oscillate back and

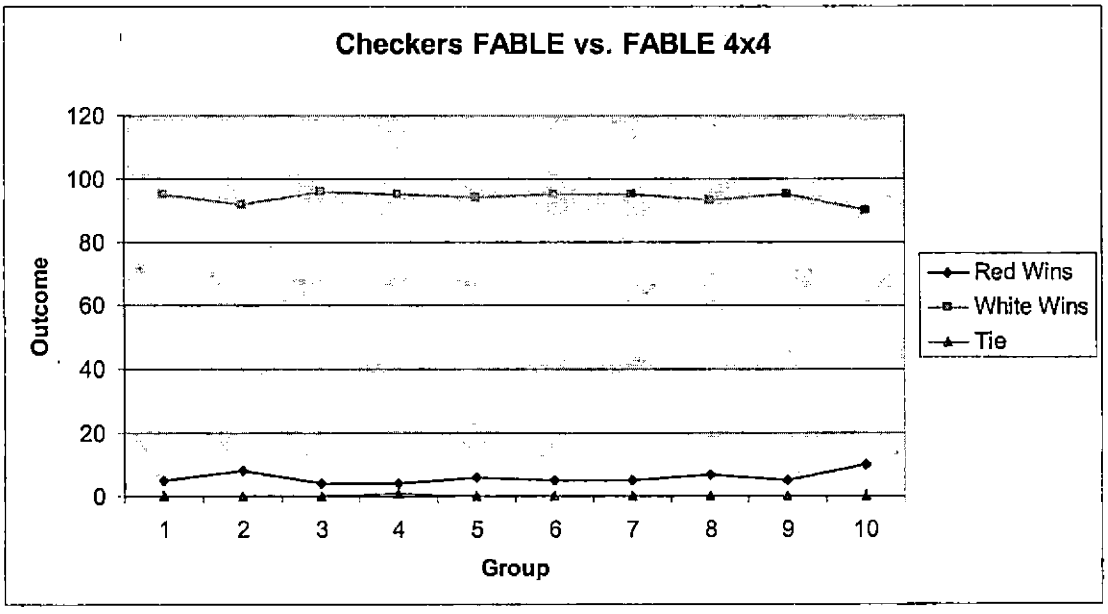

Fig. 4.25: Results of FABLE versus FABLE on Checkers 4x4.

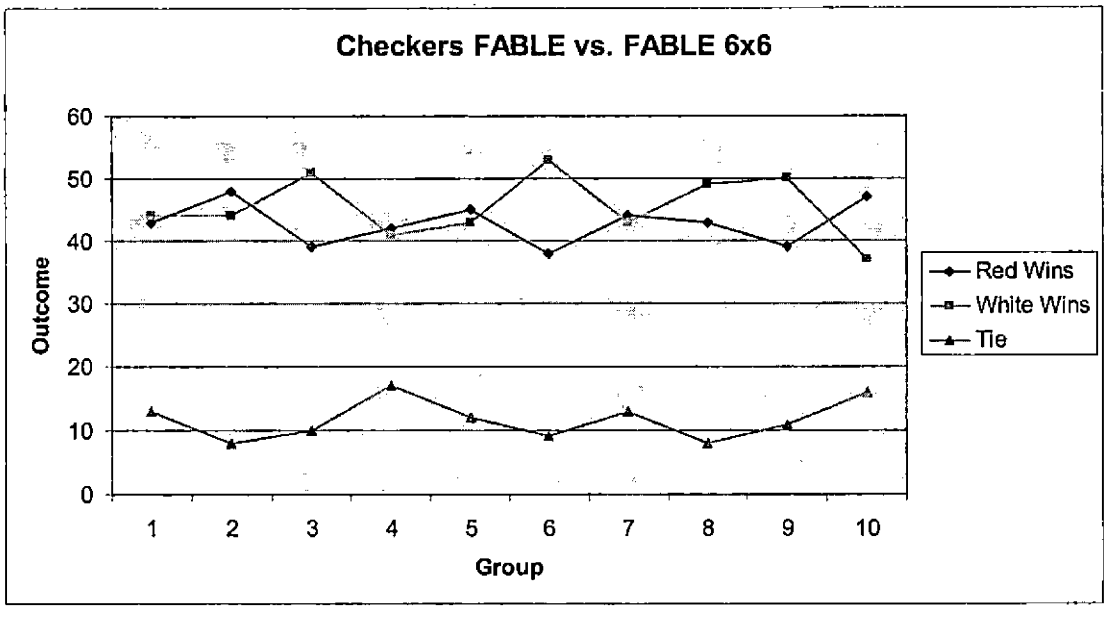

Fig. 4.26: Results of FABLE versus FABLE on Checkers 6x6.

forth between which wins more often, red or white. The difference between the result for FABLE and random though is that the results have a higher consistency. In addition, the rate at which the game ends in a tie is much higher for FABLE than for random. The rate of a tie occurring also shows signs that it may be starting to increase towards the end of the data run. If so, it may be that on a 6x6 board, the result of perfect play is a tie, just as it is on an 8x8 board.

In all cases, the data do show that FABLE is capable of learning enough about checkers within the first 100 games to make a significant change to the distribution of wins for red and white. As far as the probability of a game ending in a tie, the only case where a significant change occurred was in the case of FABLE red versus FABLE white on the 6x6 board, which showed a clearly higher rate of ties compared to any of the other test cases.

4.5 Summary

In this chapter, the actual results of running FABLE with the purpose of learning how to play 3 different games was conducted. In each case FABLE began with no prior knowledge of the problem. For each game, FABLE showed an ability to learn how to play at a level that is clearly better than random, although for each task the rate of learning took a different form. On minesweeper, the learning followed a clear saturation growth pattern. For tic-tac-toe the data appeared to follow a growth pattern in places, but it also appeared to show occasional spurts in performance followed by periods of little change. The data for checkers on the 4x4 indicate that a very rapid growth rate in performance compared to random within the first 100 games

with no clear improvement or degradation in performance afterwards. For checkers on the 6x6, the data are hard to interpret other than to say that it is at least clear that FABLE consistently outperforms random selection.

5. CONCLUSION

5.1 Introduction

This chapter will discuss the conclusions of the thesis. Ideas regarding future work will be presented, and a summary of the work will finish this thesis.

5.2 Conclusion

FABLE was created with the goal of making a learning engine where acquired knowledge is stored in the form of a FA. The FA model built by FABLE is then used to evaluate decisions with the goal of influencing outcomes towards a desired goal state. A constraint was added that this must be done in at most polynomial time. To achieve this goal, 4 heuristics were created with the purpose of building and evaluating FA models of a problem. These are the observed state heuristic, nearest neighbor heuristic, locality heuristic, and the rotationally invariant heuristic. It was hypothesized that a properly designed learning engine of this type should be able to learn to play a number of board games without any prior knowledge of the games. This is due to the fact that many board games consist of clear start states, have clear rules governing the transition from one game state to the next, and clear end states. To evaluate if FABLE is indeed capable of attaining these goals, it was tested on the board games of minesweeper, tic-tac-toe, and checkers. The tests were done by first

determining the results of random play on each game and then doing the same for FABLE. Improvement in performance was measured by comparing the actual game outcomes for FABLE with the results for random play to see if FABLE was able to influence the outcomes towards its goal state(s).

In the case of minesweeper, test cases were conducted on 3x3, 4x4, and 5x5 size grids. In each case, statistically significant improvements compared with playing at random were observed. For the 3x3 grid, it was further demonstrated that performance continued to improve until it reached the point where it was statistically indistinguishable from playing minesweeper perfectly. To determine if the same can be said for the 4x4 would require that the expected result of playing minesweeper perfectly on a 4x4 grid be calculated. For the 5x5, the performance improved on an exponential growth saturation curve which is similar to the curve for the 3x3 and the 4x4.

In the case of tic-tac-toe, FABLE was tested on 3x3, 4x4, and 5x5 boards. The rules were modified from those of standard tic-tac-toe such that instead of the game ending whenever a player completes 3 in a row, the completion of 3 in a row would instead award 1 point. The winner was determined by seeing which player had the most points at the end. For this variant of tic-tac-toe, FABLE showed performance improvements that were sufficient to distinguish from the results of random selection within the first 100 games for every test that was conducted. The improvements in performance over time for this game were harder to characterize than in minesweeper, but in general the improvements appeared to follow two basic patterns. The first was consistent upward growth similar to the saturation curve seen in minesweeper. The

second was step-wise growth where the performance level would appear to show no significant change, then make a noticeable improvement, then level off again before making another improvement. It is possible that these occasional steps in performance occurred after the FA model gains a critical transition that makes a difference in future decision making.

Checkers was tested on a 4x4 and a 6x6 board. The only clear conclusion for checkers is that FABLE shows a clear improvement over making decisions at random after less than 100 games. After those first 100 games there was no clear pattern other than that FABLE continued to exhibit better performance than making decisions at random.

In addition to the empirical data that was collected, some formulas regarding the complexity of the games being studied were derived. In particular, either the game tree size or the space complexity was analyzed for each game. For minesweeper, formulas were derived for finding the upper and lower bounds of the game tree size. For checkers, the formula developed by Jonathan Schaeffer and Robert Lake to find the space complexity of checkers on an 8x8 board [19] was generalized to NxM boards. The generalization was done with the constraint that both the value of N and the value of M must be even numbers.

The significance of the results of this thesis are that it has been established that a FABLE agent was able to learn to play each game at a level better than random play with no prior knowledge of the game. Furthermore, this learning occurred without the need to write any specialized code for the learning algorithms for any of the particular games. Since none of the code related to learning was specific to these games, it is

expected that FABLE would be able to learn to play other games without adding any new learning algorithms. The level of performance that FABLE would achieve for any specific game is likely to vary from one game to the next, just as it varied between minesweeper, tic-tac-toe, and checkers.

Overall, this was an interesting endeavor since most research conducted with the goal of automatically building FA's is focused on the ability to learn a target FA, or in other words, the goal of the research is to develop the FA itself. For FABLE, the creation of the FA was used as a means of facilitating the making of intelligent decisions while playing board games. That is, for FABLE, the learning of FA is a means to achieving an end, not the end itself. It is likely that the automatic building of FA models of problems could be used to improve outcomes compared to random decisions for other problems besides board games.

5.3 Future Work

One idea that would be beneficial for future work would be to conduct longer data runs for tic-tac-toe and checkers. The length of the data runs contained in this thesis were sufficient to establish that FABLE did learn at least some aspects of these games, but the data runs were of insufficient length to establish what the ultimate performance of FABLE would be.

What I actually plan to do for future work is to continue to find additional problems for FABLE to learn. Whenever a problem that is possible to model by an FA and has a known pattern to it is presented to FABLE and FABLE proves incapable of discovering the pattern, the question will be to ask what type of reasoning would allow

that pattern to be detected. From there, a heuristic can be devised that is designed to look for that type of pattern. As a rule, the new heuristic should be designed to look for the pattern in the most general sense possible while strictly adhering to rule that it must be tractable. In this way, new types of inference for different types of problems will be added to FABLE's capabilities.

If adding new heuristics proves to be a successful strategy for increasing the general problem solving abilities of FABLE, another problem will arise. A growing number of heuristics will require an increasing amount of processing power to run them all simultaneously. To mitigate this, a future version of FABLE could include improving the top level agent so that it can measure the quality of the results from individual heuristics. It is likely that the heuristics that are most beneficial will be different from one problem to the next. Once the heuristics are evaluated, the top level agent could be given the ability to turn individual heuristics on and off. There are many possible strategies that could be used for making decisions as to which heuristics to turn off and on. If done right, this improvement to FABLE would not add any new learning capabilities, but would allow actual performance to be affected minimally while potentially reducing total computing resources consumed significantly.

It is my conjecture that at some point, the library of available heuristics would include enough different reasoning skills that more problems that can be handled by an FA would not require creating additional heuristics to solve efficiently. Of course, there is always the possibility that there will be a classes of problems that have not been considered that require different reasoning processes to solve than those encapsulated within any of the existing heuristics, but when this occurs, it should be

possible to create a heuristic that encapsulates that reasoning process, decreasing the likelihood of encountering another gap in the future.

REFERENCES

[1] K. Fregene A, D. Kennedy B, R. Madhavan C, L. E. Parker D, and D. Wang E. A class of intelligent agents for coordinated control of outdoor terrain mapping ugvs. *Engineering Applications of Artificial Intelligence*, 18:513–531, 2005.

[2] Victor Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, University of Limburg, 1994.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. 1987.

[4] Jos L. Balczar, Josep Daz, Ricard Gavald, and Osamu Watanabe. Algorithms for learning finite automata from queries: A unified view, 1996.

[5] Kenneth Basye. An automata-based approach to robotic map learning. Technical report, AAAI Technical Report, 1992.

[6] Josh Bongard and Hod Lipson. Active coevolutionary learning of deterministic finite automata. *J. Mach. Learn. Res.*, 6:1651–1678, 2005.

[7] Lourdes Pefia Castillo. Learning minesweeper with multirelational learning. In *In Proc. of the 18th IJCAI*, pages 533–538, 2003.

[8] Orlando Cicchello and Stefan C. Kremer. Inducing grammars from sparse data sets: a survey of algorithms and results. *J. Mach. Learn. Res.*, 4:603–632, 2003.

[9] Ren Feiliang. Ebmt based on finite automata state transfer generation. Technical report, 2007.

[10] A. S. Fraenkel, M. R. Garey, D. S. Johnson, T. Schaefer, and Y. Yesha. The complexity of checkers on an n*n board. In *Proceeding of the 19th IEEE Symposium on Foundations of Computer Science*, pages 55–64. IEEE, 1978.

[11] Yoav Freund, Michael Kearns, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. Efficient learning of typical finite automata from random walks. In *In Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 315–324, 1993.

[12] E. Mark Gold. Complexity of automaton identification from given data. 1978.

[13] Hugues Juille and Jordan B. Pollack. A sampling-based heuristic for tree search applied to grammar induction. In *In Proceedings of AAAI-98*, pages 776–783. MIT Press, 1998.

[14] Richard Kaye. Minesweeper is np-complete! *Mathematical Intelligencer*, 22:9–15, 2000.

[15] K. J. Lang. Faster algorithms for finding minimal consistent dfas. Technical report, NEC Technical Report, 1999.

[16] Jose Diego Ferreira Martins, Luciana Porcher Nedel, Paulo Fernando Blauth Menezes, and Fernando Accorsi. Algorithms for learning finite automata from queries: A unified view, 2004.

[17] Arlindo L. Oliveira and Stephen Edwards. Limits of exact algorithms for inference

of minimum size finite state machines. In *In Proceedings of the Seventh Workshop on Algorithmic Learning Theory, number 1160 in Lecture Notes in Artificial Intelligence*, volume 1160, pages 59–66. Springer-Verlag, 1996.

[18] Jonathan Schaeffer, Neil Burch, Yngvi Bjrnsson, Akihiro Kishimoto, Martin Mller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. 317(5844):1518–1522, 2007.

[19] Jonathan Schaeffer and Robert Lake. Solving the game of checkers. *Games of No Chance*, 29:119–133, 1996.

[20] ACF Website Team. Acf online games archive, 2007.

[21] ACF Website Team. Rules of play and laws for standard american checkers, 2007.

[22] T. Yokomori. Learning non-deterministic finite automata from queries and counterexamples. In *Machine Intelligence*, pages 169–189. Univ. Press, 1993.